CS 255, Spring 2014, SJSU

Graphs: Basics, Traversing algorithms

Fernando Lobo

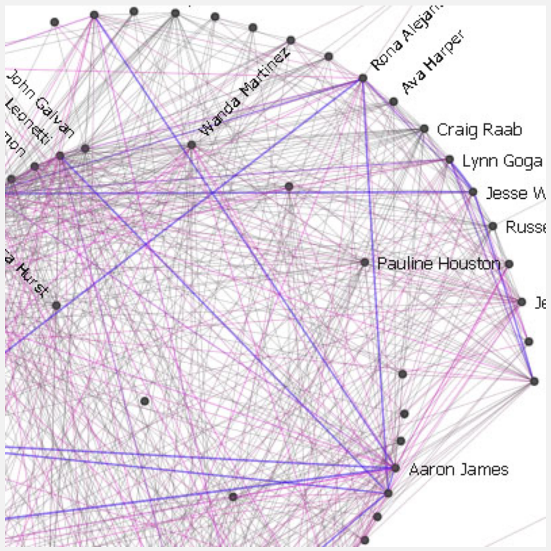
Graph basics

- Graphs are fundamental data structures in CS, just like trees and lists.
- Many interesting algorithms over graphs with many practical applications.

A graph is a network

- Computer network.
- Road network.
- Electrical netwrok.
- Social network.
- The web.
- A network of whatever you want.

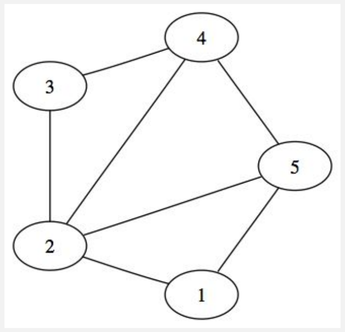
Facebook



Graphs

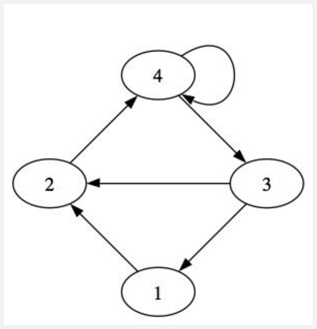
- A graph can be directed or undirected.
- Notation: $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges between vertices.

Example of an undirected graph



- Edge direction doesn't matter: $(u, v) = (v, u)$.
- $V = \{1, 2, 3, 4, 5\}$
- $E = \{(1, 2), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5)\}$

Example of a directed graph



- Edge direction matters: $(u, v) \neq (v, u)$.
- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (2, 4), (3, 1), (3, 2), (4, 3), (4, 4)\}$

Representation

Two main representations:

1. Adjacency matrix.
2. Array of adjacency lists.

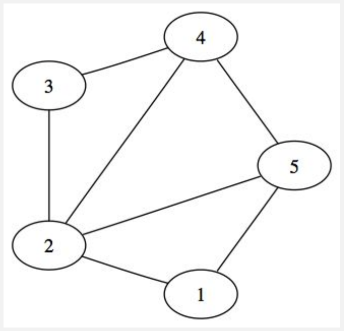
Adjacency matrix

- Build a matrix A of size $|V| \cdot |V|$ such that:

$$A_{ij} = \begin{cases} 1 & \text{, if } (i,j) \in E \\ 0 & \text{, otherwise} \end{cases}$$

Undirected graph

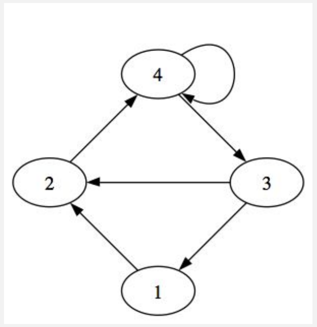
$$\begin{array}{c|c|c|c|c|c|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 2 & 1 & 0 & 1 & 1 & 1 \\ \hline 3 & 0 & 1 & 0 & 1 & 0 \\ \hline 4 & 0 & 1 & 1 & 0 & 1 \\ \hline 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$



Matrix is symmetric: $A[i,j] = A[j,i]$.

Directed graph

$$\begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ \hline 2 & 0 & 0 & 0 & 1 \\ \hline 3 & 1 & 1 & 0 & 0 \\ \hline 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

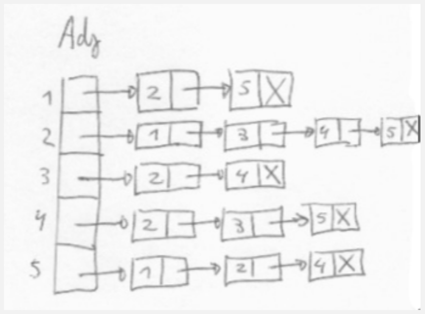


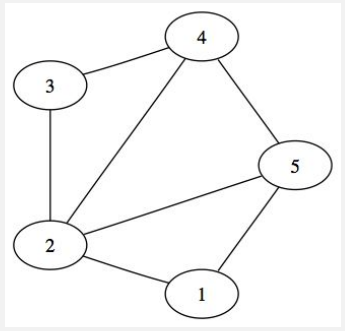
Matrix is not symmetric.

Array of adjacency lists

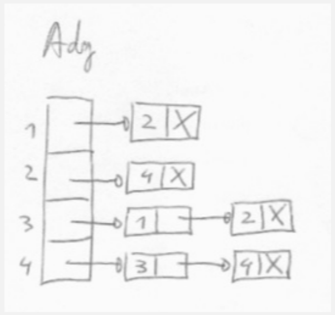
- The graph is represented by an array of lists, one for each vertex.
- The list for vertex v contains all the vertices adjacent to v.

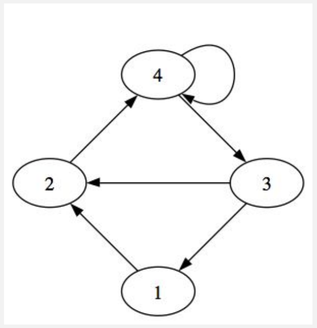
Undirected graph





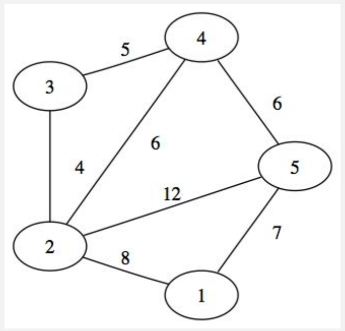
Directed graph





A graph can have weights

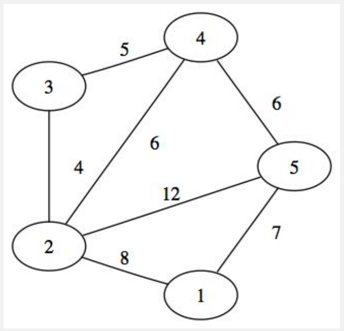
- Many times we associate weights with edges. They can represent costs, distances, and so on.



Adjacency matrix representation

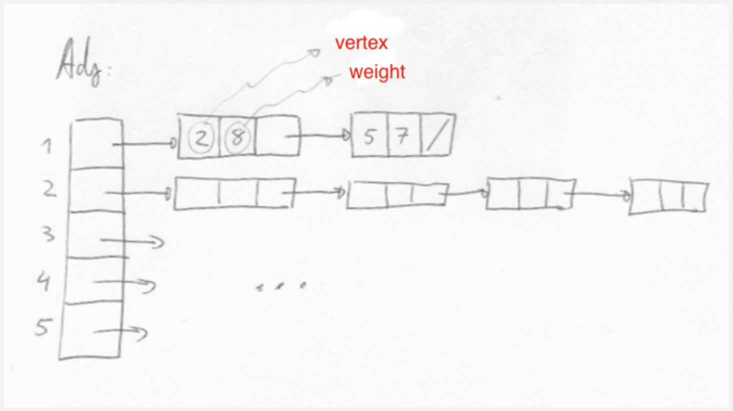
- With an adjacency matrix we keep a weight (a real number) instead of a bit. Use a special number, different from any possible weight, to denote an edge absence.

	1	2	3	4	5
1	-1	8	-1	-1	7
2	8	-1	4	6	12
3	-1	4	-1	5	-1
4	-1	6	5	-1	6
5	7	12	-1	6	-1



Adjacency list representation

- With adjacency lists we keep for each element in the list, the vertex and the corresponding weight.



Adjacency matrix vs. Array of adjacency lists

- What's the best representation?
- Answer: It depends.
  - Matrix is better for dense graphs, when the number of edges = $\Theta(|V|^2)$.
  - Adjacency lists is better for sparse graphs $\rightarrow$ because in that case most matrix entries will be empty.

Running time analysis

- We've been looking at the running time of algorithms in terms of a parameter N which reflects the size of the input.
- With graph algorithms we usually analyze the running time in terms of two parameters: $|V|$ and $|E|$.
- In asymptotic notation we usually use V and E instead of $|V|$ and $|E|$. That's an abuse of notation. We just use them to make the expressions look simpler.

Space complexity

- Adjacency matrix: $\Theta(V^2)$
- Array of adjacency lists: $\Theta(V + E)$

Time complexity

- Depends on the operations.
- Two fundamental operations:

  op1: check if $(u, v) \in E$.

  op2: list all the vertices that adjacent to a given vertex u.

Time complexity

	op1	op2
Adjacency matrix	$\Theta(1)$	$\Theta(V)$
Array of adjacency lists	$\Theta(\text{degree}(u))$	$\Theta(\text{degree}(u))$

- degree(u) is the number of vertices adjacent to u.
- Adjacency matrix representation is better for op1. Array of adjacency lists is better for op2.

In practice

- Dense graphs $\rightarrow$ Adjacency matrix.
- Sparse graphs $\rightarrow$ Array of adjacency lists.
- For most applications, graphs are sparse.
- We shall use the array of adjacency lists representation in the majority of algorithms that we will see.

Side note

- In the examples, the vertices are identified by integers.
- In practice, vertices can have names and other associated information.
- It is necessary to have a table that maps those names (and extra info) to the vertex identifiers.

Path and connectivity in graphs

We shall now see algorithms that

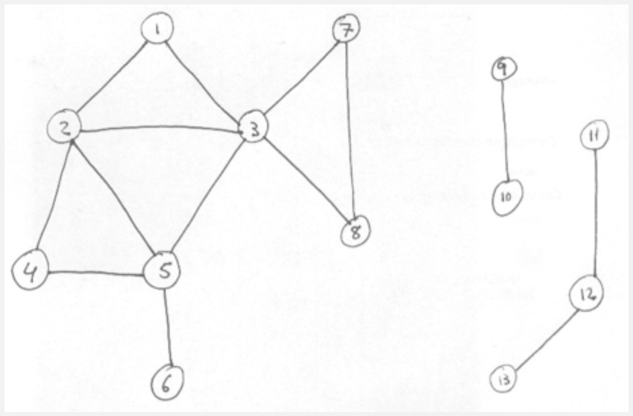
- visit the vertices of a graph in a systematic manner.
- determine if there is a path between two vertices.

Path and connectivity in graphs

Two fundamental algorithms:

- BFS $\rightarrow$ Breadth-First Search.
- DFS $\rightarrow$ Depth-First Search.

BFS – Breadth-First Search

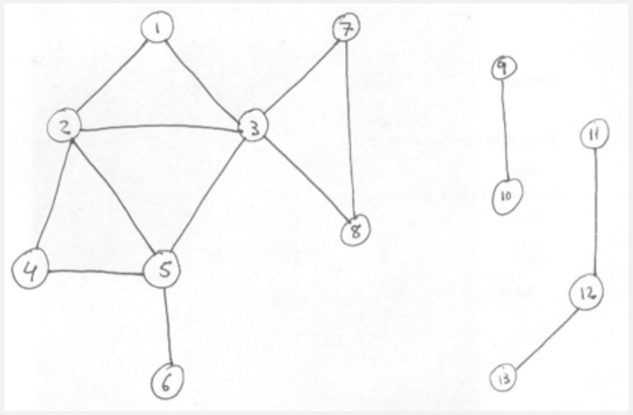


Given a starting vertex s, BFS visits the remaining vertices by layers: L_1, L_2, L_3, ..., as if it was a tsunami.

BFS

- L_1: vertices that are at a distance 1 from vertex s.
- L_2: vertices that are at a distance 2 from vertex s.
- L_3: vertices that are at a distance 3 from vertex s.
- ...

BFS



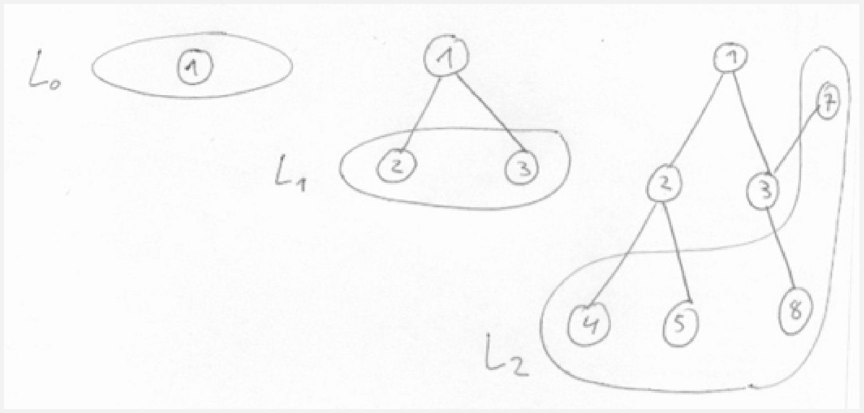
If s is vertex 1, then:

- L_0: {1}
- L_1: {2, 3}
- L_2: {4, 5, 7, 8}
- L_3: {6}

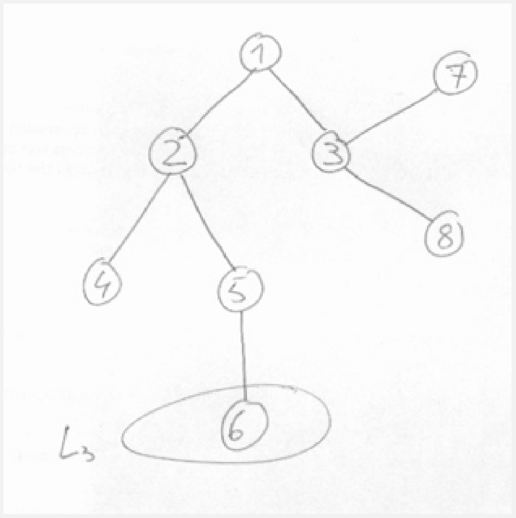
BFS

- Formally:
- $L_0 = \{s\}$
- Assuming we have $L_0, L_1, L_2, \ldots, L_j$, then L_{j+1} is the set of vertices that to not belong to any of the previous layers, and which have an edge to a vertex in layer L_j.

L_0, L_1, L_2



L_3



BFS

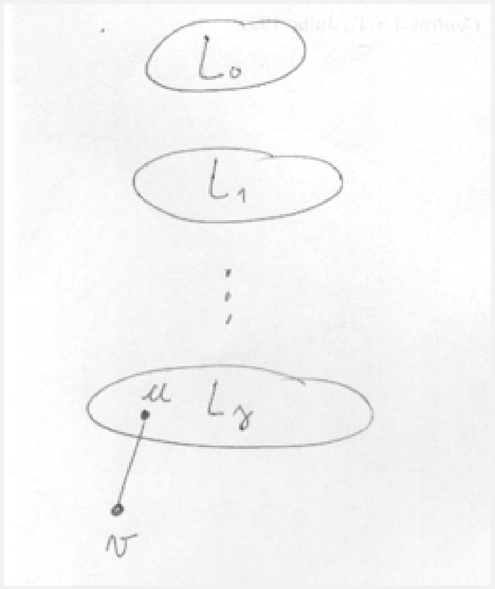
- L_j contains all the nodes which are at a distance j from s.
- If a vertex doesn't show up in any layer, then there's no path from s to that vertex.

In summary

BFS gives us the set of vertices for which there is a path from a starting vertex s, and it also computes the shortest path from s to those vertices.

BFS Tree

- BFS produces a BFS Tree with vertex s as root.
- The BFS Tree contains the vertices for which there is a path from s.
- For each vertex v ($\neq s$), consider the moment where v is "discovered" by the BFS algorithm. This happens when $u \in L_j$ is being visited and we check an edge (u, v) where v has not been visited yet.
- At that moment we add (u, v) to the BFS Tree (u becomes parent of v).



BFS implementation

- Uses a FIFO queue to store the vertices on “top of the tsunami wave”.
- We use a color attribute (WHITE, GRAY, BLACK) for each vertex.
- At the beginning all vertices are WHITE.
- They stop being WHITE when they are discovered for the first time. The GRAY vertices are those “on top of the tsunami wave”. The BLACK vertices were already swept by the tsunami.

BFS implementation

- Input: a graph G and a starting vertex s.
- Output:
  - $v.d \rightarrow$ distance from s to v, $\forall_v \in V$.
    (distance is the minimum number of edges from s to v).
  - $v.\pi \rightarrow$ parent of v in the BFS Tree, $\forall_v \in V$.

Pseudocode

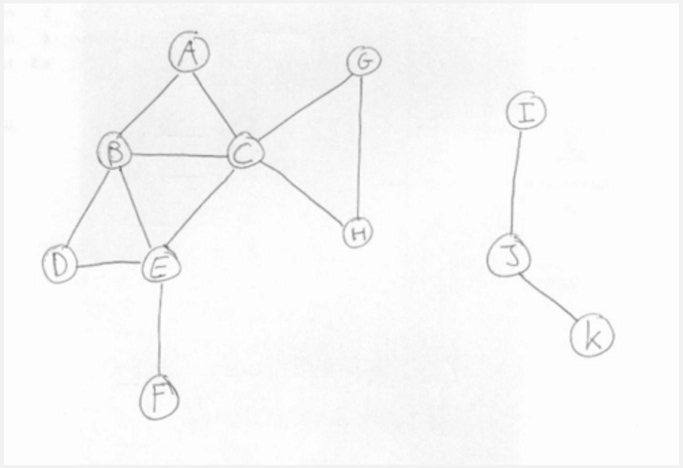
```
BFS($G, s$)
    for each $u \in G.V - \{s\}$
        $u.color$ = WHITE
        $u.d = \infty$
        $u.\pi$ = NIL
    $s.color$ = GRAY
    $s.d = 0$
    $s.\pi$ = NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    ⋮
```

Continues next page.

Pseudocode (cont.)

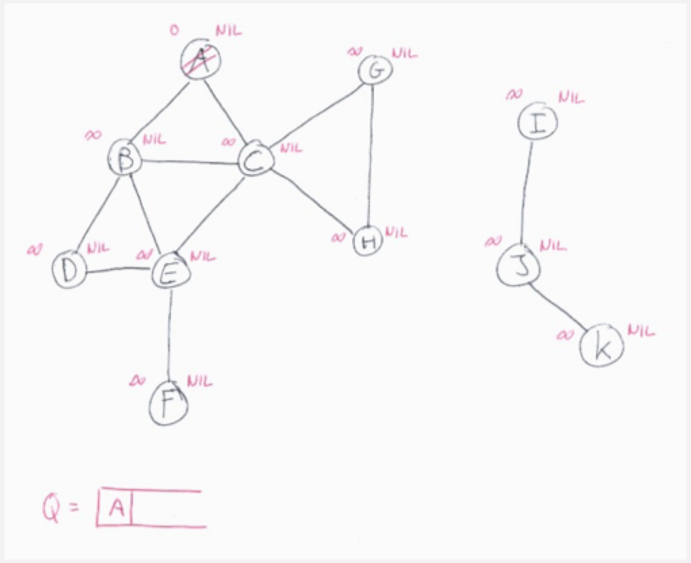
```
BFS($G, s$)
    ⋮
    while $Q \neq \emptyset$
        $u$ = DEQUEUE($Q$)
        for each $v \in G.Adj[u]$
            if $v.color$ == WHITE
                $v.color$ = GRAY
                $v.d = u.d + 1$
                $v.\pi = u$
                ENQUEUE($Q, v$)
        $u.color$ = BLACK
```

Example: *s* is node A

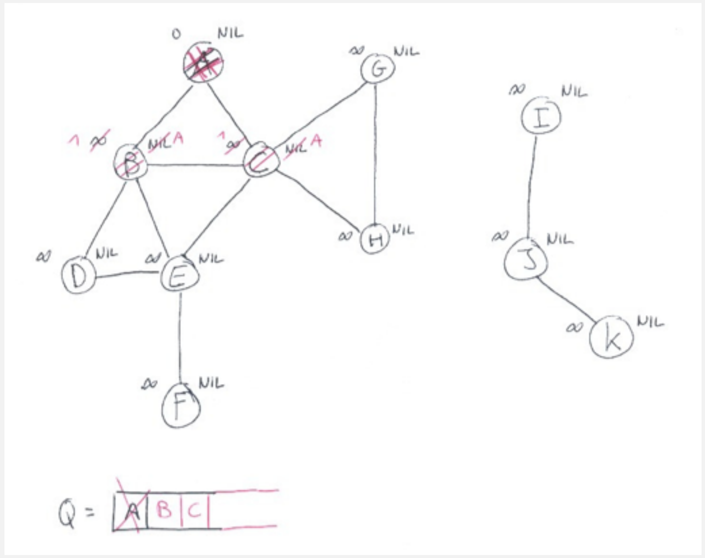


Notation: ◯ WHITE ▨ GRAY ⊗ BLACK

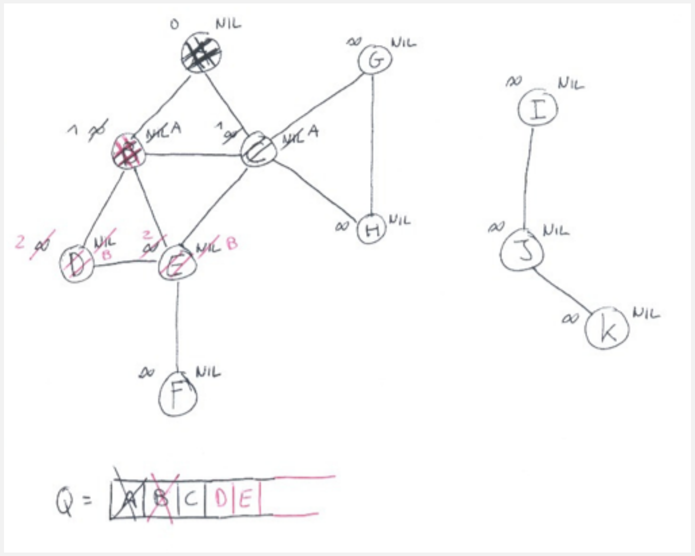
Example



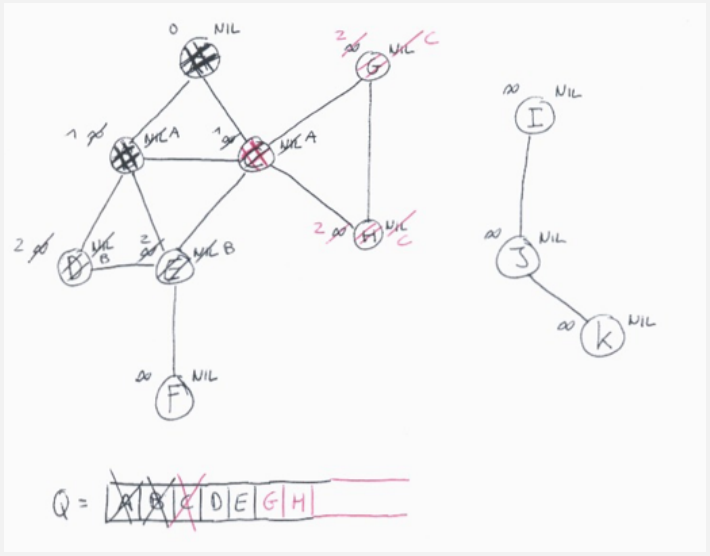
Example



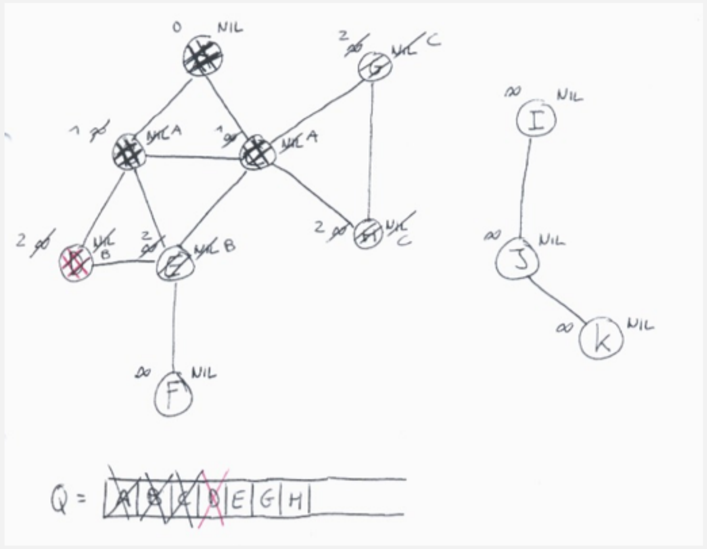
Example



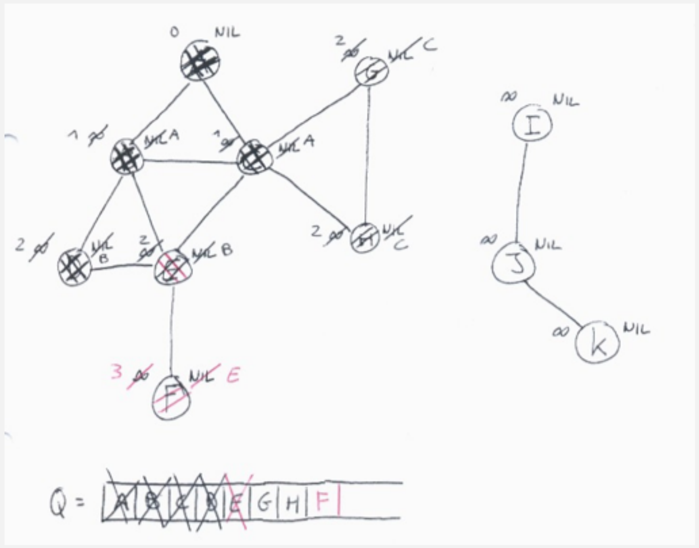
Example



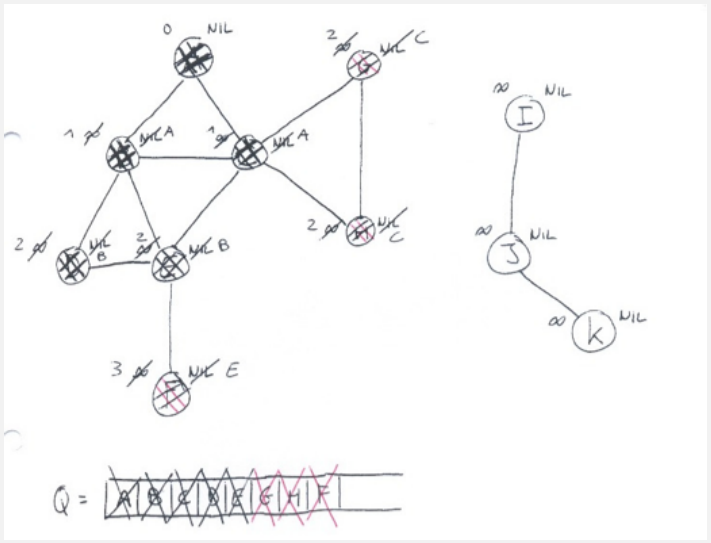
Example



Example



Example



BFS implementation

- If a graph has more than one connected component, BFS does not visit all the vertices. In the example, vertices I, J, K are not visited.
- After running BFS we can obtain the BFS Tree from the information stored in the π attribute of the vertices.
- The set of edges $\{(v.\pi, v) : v \neq s\}$ gives us the BFS Tree.
- In the example, {(A,B),(A,C),(B,D),(B,E),(C,G),(C,H),(E,F)}.

Running time of BFS

- Each vertex enters the queue Q at most one time. It also only leaves the queue at most one time.
- ENQUEUE and DEQUEUE = $\Theta(1)$.
- Therefore, the operations on the queue = $O(V)$.
- The adjacency list of each vertex is only traversed when the vertex gets out of the queue (at most one time). Since the sum of the sizes of all adjacency lists = $\Theta(E)$, we spend $O(E)$ to traverse the lists.
- Initialization cost = $\Theta(V)$.
- Total running time = $O(V + E)$.

Shortest path

- After running BFS, we can determine the shortest path between s and another vertex v.

```
PRINT-PATH($G, s, v$)
  if $v == s$
      print $s$
  elseif $v.\pi ==$ NIL
      print "No path from" $s$ "to" $v$
  else PRINT-PATH($G, s, v.\pi$)
      print $v$
```

Connected components

- The set of vertices discovered by BFS is the set of vertices that are connected to s by some path.
- Such a set (call it R) is a connected component of G that contains s.
- To check if there is a path from s to t, it suffices to check if $t \in R$.
- If we run BFS again starting from an unvisited vertex, we will obtain another connected component.
- We can easily change the BFS pseudocode to obtain all the connected components of a graph.

DFS – Depth-First Search

- As soon as a vertex is discovered, we start exploring its descendants. When all descendants have been explored, we backtrack.
- As opposed to BFS, the search restarts from a new vertex. In other words, all the vertices of the graph are visited, even if the graph is not connected.
- Instead of a BFS Tree, produces a DFS Forest (several trees, one for each connected component.)

DFS implementation

- Use a stack instead of a FIFO queue (No need of stack if we implement a recursive algorithm.)
- Maintain two *timestamps* for each vertex:
  - $v.d \rightarrow$ discovery time, instant when v is discovered.
  - $v.f \rightarrow$ finish time, instant when all the descendants of v have been explored.
- These timestamps are distinct integers from 1 to $2|V|$
- $v.d < v.f \ , \ \forall_v$

DFS implementation

Also uses a color attribute for each vertex:

- WHITE $\rightarrow$ not discovered.
- GRAY $\rightarrow$ discovered but not finished (there's still descendants to explore).
- BLACK $\rightarrow$ finished.

For a given vertex v:

- v is WHITE between timesteps 1 and $v.d$
- v is GRAY between timesteps $v.d$ and $v.f$
- v is BLACK between timesteps $v.f$ and $2|V|$

Pseudocode

```
DFS($G$)
    for each $u \in G.V$
        $u.color$ = WHITE
        $u.\pi$ = NIL
    $time = 0$
    for each $u \in G.V$
        if $u.color$ == WHITE
            DFS-VISIT($u$)
```

Pseudocode

```
DFS-VISIT($u$)
    $u.color = \text{GRAY}$          // $u$ has just been discovered
    $time = time + 1$
    $u.d = time$
    for each $v \in G.Adj[u]$        // explore edge $(u, v)$
        if $v.color == \text{WHITE}$
            $v.\pi = u$
            DFS-VISIT($v$)
    // finished exploring the descendants of $u$
    $u.color = \text{BLACK}$
    $time = time + 1$
    $u.f = time$
```

Running time for DFS

- Running time for DFS = $\Theta(V + E)$.
- Θ and not O as in BFS, because here all vertices and all edges are processed.
- $\Theta(V)$ for initialization.
- $\Theta(E)$ because for each vertex u, we process $|Adj[u]|$ edges.

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

Example

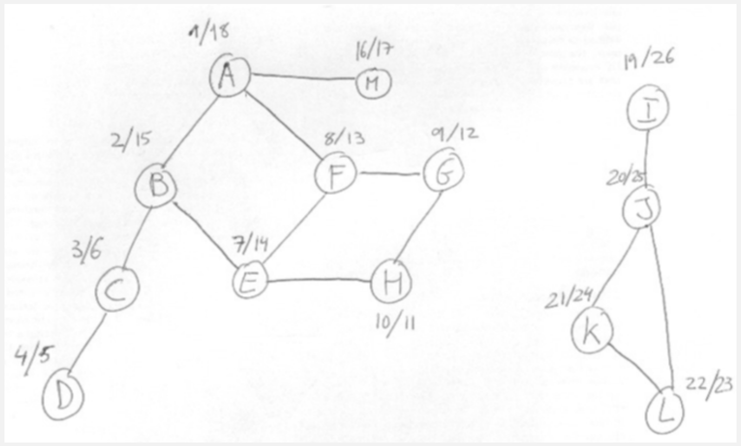


Figure notation: $d/f \rightarrow$ discovery time / finish time.

Edge classification

- Tree edge: if it belongs to the DFS Forest. (u, v) if v is discovered while exploring (u, v).
- Back edge: (u, v) if v is a predecessor of u in a DFS Tree.
- Forward edge: (u, v) if v is a descendant of u is a DFS Tree, and is not a Tree edge.
- Cross edge: all the other edges of the graph.

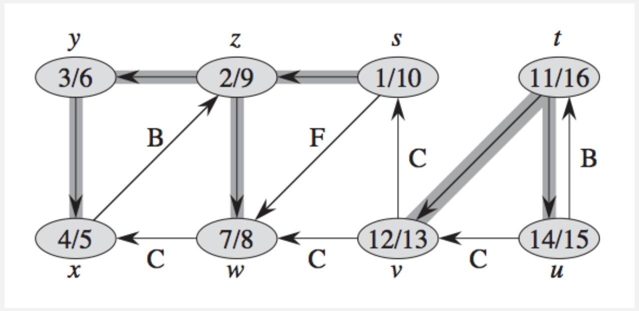
An undirected graph only has Tree edges and Back edges.

Edge classification

Can classify an edge (u, v) while exploring it in the **for** loop of DFS-VISIT

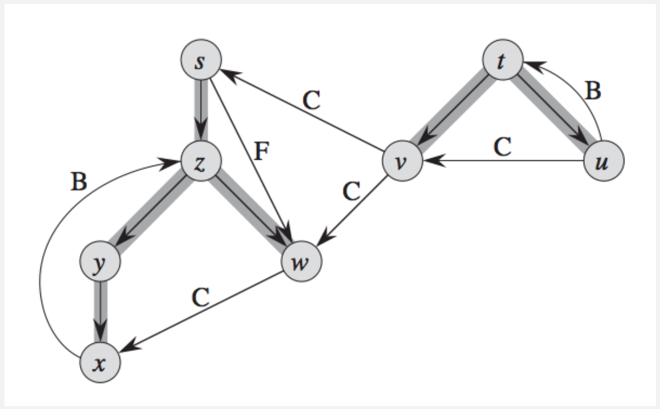
- if v is WHITE, then (u, v) is a Tree edge.
- if v is GRAY, then (u, v) is a Back edge.
- if v is BLACK and $u.d < v.d$, then (u, v) is a Forward edge.
- if v is BLACK and $u.d > v.d$, then (u, v) is a Cross edge.

Example (taken from CLRS textbook)



Notation: B $\rightarrow$ back edge, F $\rightarrow$ forward edge, C $\rightarrow$ cross edge.

Redrawing the graph (taken from CLRS textbook)



Tree edges and forward edges from top to bottom.
Back edges point to predecessors.

Parenthesis theorem (see proof in the textbook)

Given two vertices, u and v, one of the following 3 things occur:

1. $u.d < u.f < v.d < v.f$ or
   $v.d < v.f < u.d < u.f$
   (u is not a descendant of v, v is not a descendant of u).
2. $u.d < v.d < v.f < u.f$
   (v is a descendant of u).
3. $v.d < u.d < u.f < v.f$
   (u is a descendant of v).

Discovery and finishing times have parenthesis structure.

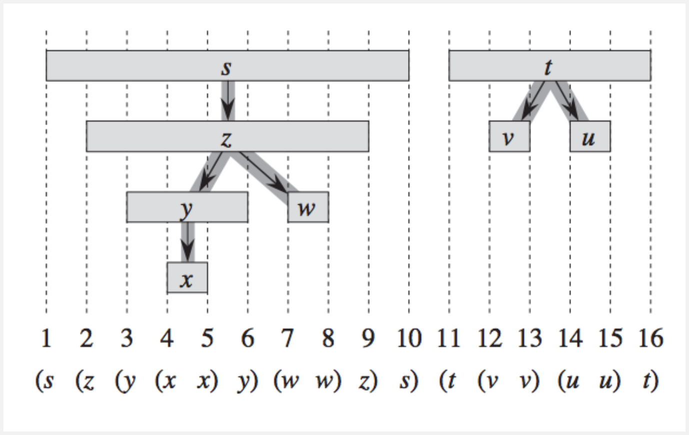
Parenthesis theorem

Let

$$\begin{cases} u.d & \rightarrow & (\\ u.f & \rightarrow &) \\ v.d & \rightarrow & [\\ v.f & \rightarrow &] \end{cases}$$

- Case 1: ( ) [ ] or [ ] ( )
- Case 2: ([ ])
- Case 3: [( )]
- Things like ([ )] or [( ]) never occur in DFS.

Parenthesis theorem (example)



This example is with graph we've just seen (taken from CLRS).

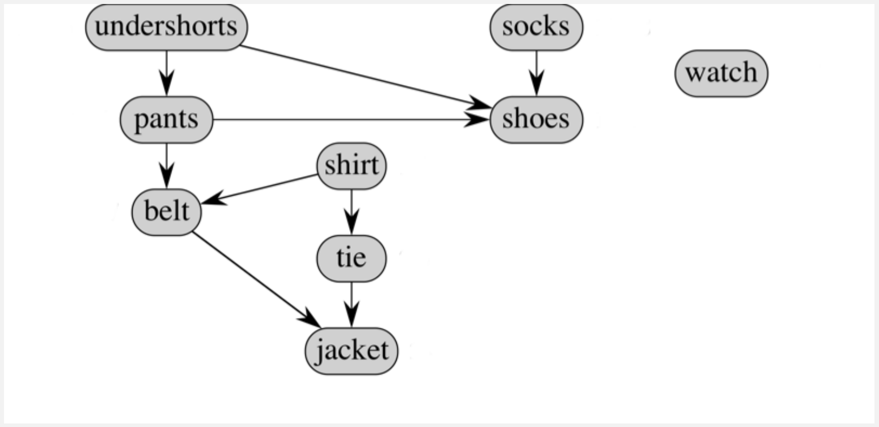
White-Path Theorem

- v is a descendant of u, if an only if, at time step $u.d$, there is a path $u \rightsquigarrow v$ solely made from WHITE vertices (with an exception for u which has just turned GRAY).

Directed Acyclic Graphs (DAGs)

- Directed graphs without cycles (loops).
- Often used to express dependencies.

Example: getting dressed (from textbook)



Topological sort

- Aplicable to DAGs
- It's an ordering of the vertices of a DAG in such a way that if $(u, v) \in E$, then u appears before v in the ordering.

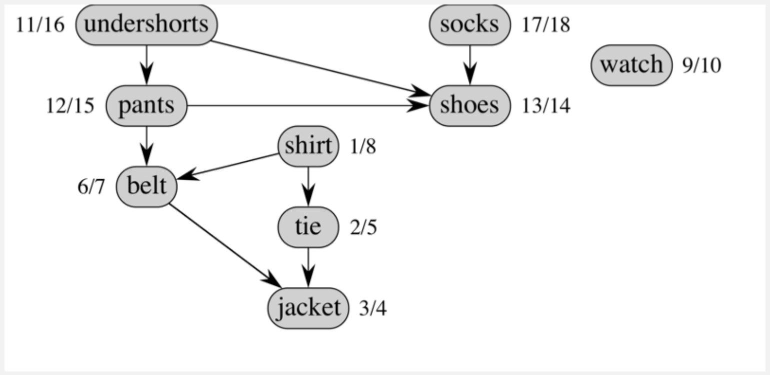
```
TOPOLOGICAL-SORT(G)
1  Call DFS(G) to obtain $v.f$ , $\forall_v$
2  As soon as a vertex finishes, insert it at the head of a linked list.
3  Return the linked list of vertices.
```

Note: Steps 2 and 3 give us the vertices of the graph in decreasing order of finish time.

Running time?

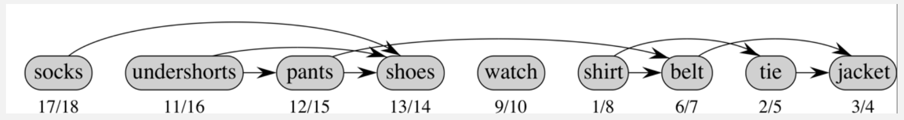
- $\Theta(V + E)$
- Same as DFS.
- Inserting in a linked list is $\Theta(1)$.

Example



T.S. = socks, undershorts, pants, shows, watch, shirt, belt, tie, jacket.

Redrawing the DAG



The graph can be redrawn with all the edges going from left to right.

Lemma

- A directed graph G has no cycles if and only if the DFS algorithm does not create a back edge.
- See proof in textbook.

Algorithm correctness

- Let $G = (V, E)$ be a DAG. We must prove that after running DFS on G,

$$v.f < u.f \ , \ \forall_{(u,v)} \in E$$

- When DFS explores (u, v), what are the colors of u and v?
  - u is GRAY.
  - v is GRAY? No.
    Otherwise v would be a predecessor of u, and (u, v) would give a cycle, which would contradict the fact that G is a DAG.
  - v is WHITE?
    If yes then v is a descendant of u, which would imply $v.f < u.f$
  - v is BLACK?
    If yes, then v has already finished. Since we are exploring (u, v), we still didn't finish u. Therefore, $v.f < u.f$ $\square$