Graph Algorithms

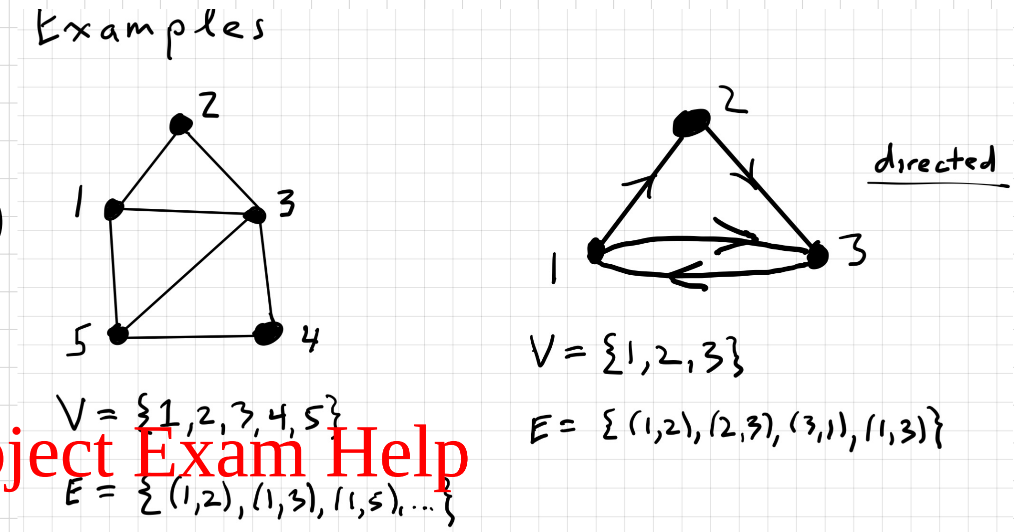
Graph $G = (V, E)$

V – vertices (nodes) $\quad |V| = n$

$E \subseteq V \times V$ – edges $\quad |E| = m$

edges can be undirected (unordered pairs)

or directed (ordered pairs)

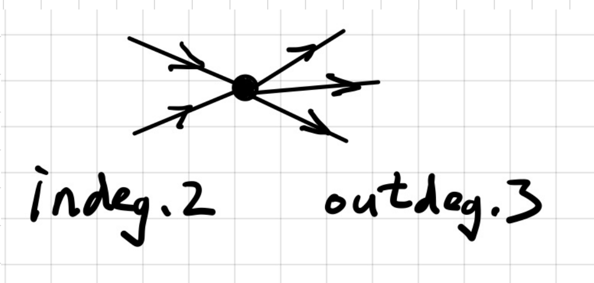


Basic Notions

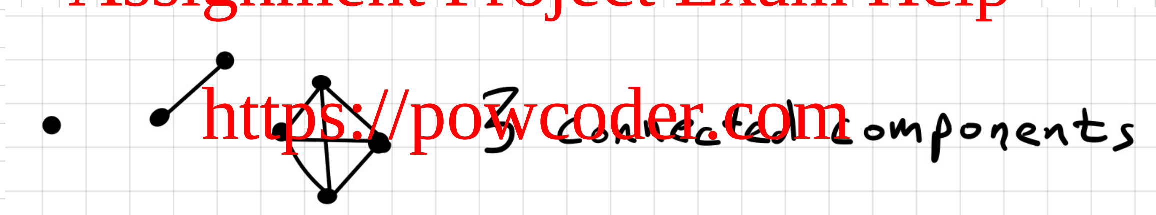
- $u, v \in V$ are <u>adjacent</u> or <u>neighbours</u> if $(u, v) \in E$

- $v \in V$ is <u>incident</u> to $e \in E$ if  $e = (v, u)$

- $\deg(v)$ = # incident edges
- for directed graph <u>indegree</u>(v), <u>outdegree</u>(v)



- a <u>path</u> is a sequence of vertices $v_1, v_2, \ldots, v_k$ s.t. $(v_i, v_{i+1}) \in E, i = 1, \ldots, k-1$
  a <u>simple</u> path does not repeat vertices
- a <u>cycle</u> is a path that starts and ends at the same vertex. <u>simple cycle</u> – no repeats
  CAUTION: Some sources use "path" to mean a simple path
- a <u>tree</u> is a connected (undirected) graph without cycles
- an undirected graph is <u>connected</u> if every $u, v \in V$ are joined by a path
- <u>connected component</u> of a graph = maximal connected subgraph



History: Euler, Königsberg bridge problem 1735. W Seven Bridges of Königsberg

Applications — many!

- networks: wireless, transportation, social
- web pages, game configurations etc.
- W Graph Theory

Storing Graphs

In practice, vertices and edges may have names or other associated information, but our algorithms will be for abstract graphs.

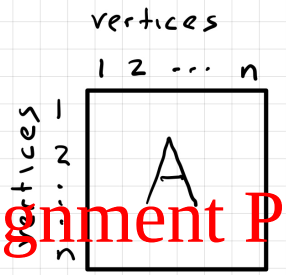
Assume vertices are $\{1, 2, \ldots, n\}$ (sometimes write $v_1, \ldots, v_n$ or use letters)

Two basic ways to store a graph:

Adjacency matrix

$n \times n$ matrix

space $O(n^2)$

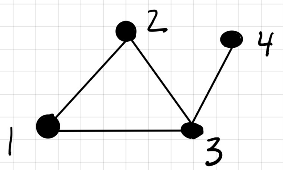


$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency lists for every vertex u, store a linked list of its (forward) neighbours, i.e., vertices v such that $(u, v) \in E$

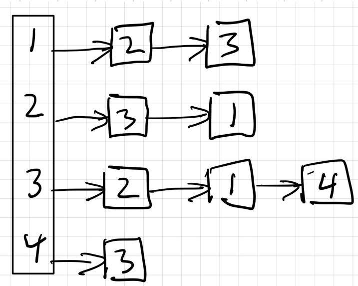
space $O(n + m)$

Examples



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

for an undirected graph $A[i,j] = A[j,i]$



For an undirected graph, every edge "appears" twice, e.g., $(2, 3)$ is in 2's list and 3's list.

More examples in CLRS or http://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf

Ex. Do an example of a directed graph.

Basic operations:

	adjacency matrix	adjacency lists
list v's neighbours	$\Theta(n)$	$\Theta(1 + \deg(v))$
list all edges	$\Theta(n^2)$	$\Theta(n + m)$
is $(u, v) \in E$	$\Theta(1)$	$O(1 + \deg(u))$
space	$\Theta(n^2)$	$\Theta(n + m)$

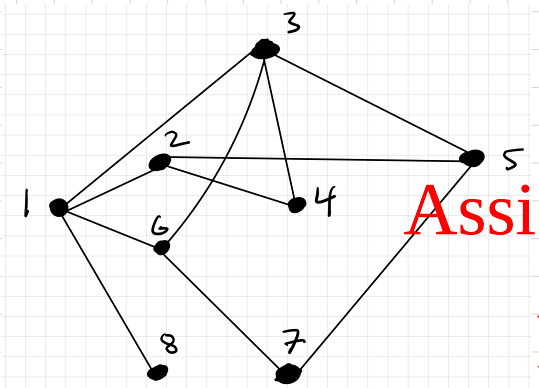
our algorithms will only need these

For algorithms in this course, we'll use adjacency lists.

<u>Exploring Graphs</u> – visit all nodes, or all nodes reachable from some "source" further – find shortest paths, connected components.

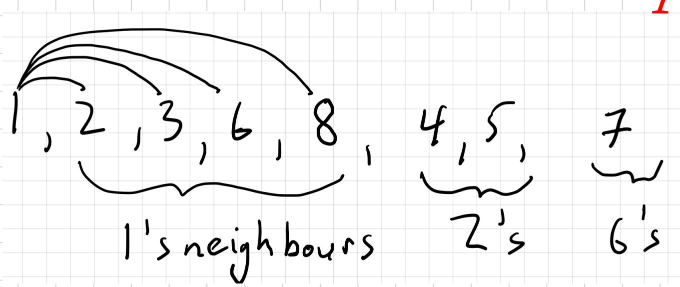
Breadth First / Depth First Search

BFS

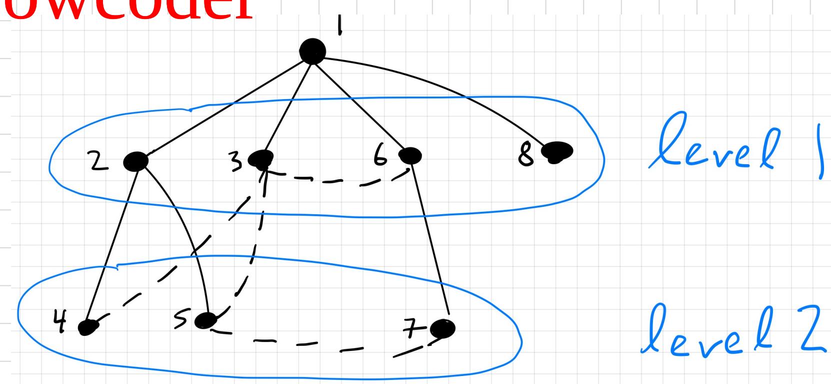


Cautious search: check everything one edge away, then two, etc.

order in which vertices are discovered



BFS tree



Use a queue to store vertices that have been <u>discovered</u> but must still be <u>explored</u>.
Vertices are marked: undiscovered $\rightarrow$ discovered.

Explore(v)
  for each neighbour u of v **do**
    if mark(u) = undiscovered **then**
      mark(u) := discovered   parent(u):= v ; level(u) := level(v)+1
      add u to Queue
    fi
  od

BFS
  initialize: mark all vertices undiscovered
  pick initial vertex v_0 parent(v_0) := ∅; level(v_0) := 0
  add v_0 to Queue
  mark(v_0) := discovered
  while Queue not empty **do**
    v := remove from Queue
    Explore(v)
  od

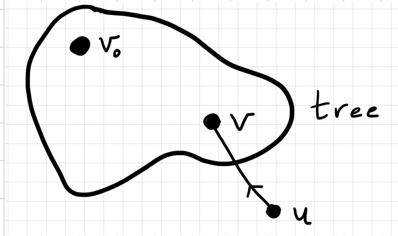
Also useful to store parent and level. See blue additions above.

BFS takes $O(n+m)$ time — we explore each vertex once and check all incident edges. Time is $O(n + \sum_v \deg(v)) = O(n+m)$

Note: $\sum_v \deg(v) = 2m$ because we count each edge twice.

Properties of BFS

- the parent pointers create a directed tree (because each addition adds a new vertex u, with parent v in the tree)



- u is connected to v_0 if and only if BFS from v_0 reaches u.

Stronger: Lemma: The level of a vertex v = length of shorted path from v_0 to v.
Proved via 2 claims:
Claim 1 v in level $i \Rightarrow$ there is a path v_0 to v of i edges.
Claim 2 v in level $i \Rightarrow$ every path v_0 to v has $\geq i$ edges.

Proof of claim 1 by induction on i, the level. Basis $i = 0$: $v = v_0$, the root of the tree. Induction step: v in level $i \Rightarrow$ parent(v) in level $i-1 \xRightarrow[\text{hyp.}]{\text{ind.}}$ there is a path v_0 to parent(v) of $i-1$ edges. Adding edge (parent$(v), v)$ gives path v_0 to v of i edges $\square$

To prove claim 2 we will prove: if there is a path v_0 to v of j edges then v is in level $\leq j$. Proof by induction on j. Basis $j = 0$: must have $v = v_0$, which is in level 0. Induction step. Let u be vertex before v in path. There is a path v_0 to u of $j-1$ edges. By induction u is in level $\leq j-1$. So one edge (u, v) goes to level $\leq j$. $\square$

Consequences:

1. BFS from v_0 finds the connected component of v_0.
2. BFS finds shorted paths (# edges) from v_0

Exercises:

- Enhance BFS to find all connected components in time $O(n + m)$.
- Use BFS to find if a connected graph has a cycle.
- Prove that if $(u, v) \in E$ then level(u), level(v) differ by 0 or 1.