Graph Algorithms

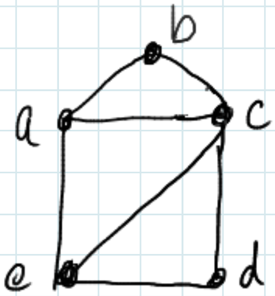
Graph $G = (V, E)$

V - vertices (nodes) $|V| = n$

$E \subseteq V \times V$ - edges $|E| = m$

edges can be undirected (unordered pairs)
or directed (ordered pairs).

Examples



directed

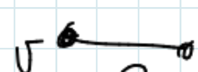
$V = \{a, b, c\}$

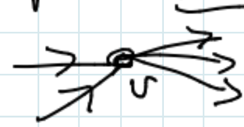
$E = \{(a,b), (b,c), (a,c), (c,a)\}$

$V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (a,e), \ldots\}$

Basic Notions

- $u, v \in V$ are adjacent or neighbours if $(u,v) \in E$
- $u \in V$ is incident to $e \in E$ if  $e = (v, u)$
- $\deg(v)$ = # incident edges
- for directed graph indegree(v), outdegree(v)



indeg. 2 outdeg. 3

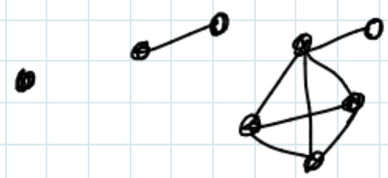
- a <u>path</u> is a sequence of vertices
  $v_1, v_2, \ldots, v_k$ s.t. $(v_i, v_{i+1}) \in E$ $i = 1 \ldots k-1$
  a <u>simple</u> path does not repeat vertices.

- a <u>cycle</u> is a path that starts and ends at the same vertex.

- a <u>tree</u> is a connected undirected graph without cycles

- a 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



3 connected components.

History: Euler, Konigsberg bridge problem 1735

Applications – many!

- networks : wireless, transportation, social
- web pages, game configurations etc

Storing Graphs

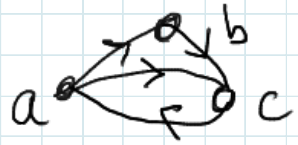
- adjacency matrix $A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$

  $O(n^2)$ space, even if the graph is <u>sparse</u>, $|E| \ll n^2$

  But a query "is (i,j) an edge" can be answered in $O(1)$

- adjacency lists

  For each vertex v, store linked list of v's neighbours



a: b, c
b: c
c: a

For G directed, every edge appears in 1 list.
For G undirected, every edge appears in 2 lists.

  $O(n+m)$ space

  A query "is (i,j) an edge?" requires traversing i's adjacency list. $O(n)$ worst case

Sometimes graphs are stored implicitly, e.g. nodes may represent configurations in a chess game. Generate nodes as you search configuration space.

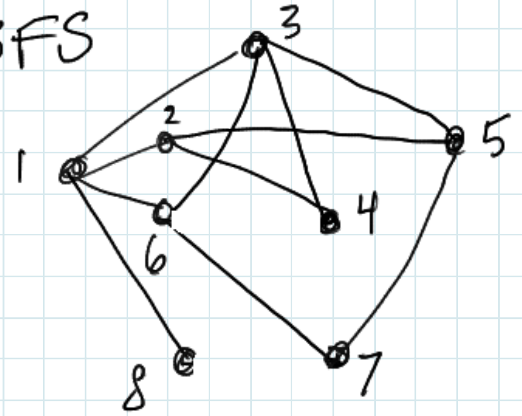
Can use hash table of adjacency lists to get space $O(n+m)$ and $O(1)$ test for edge.

<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

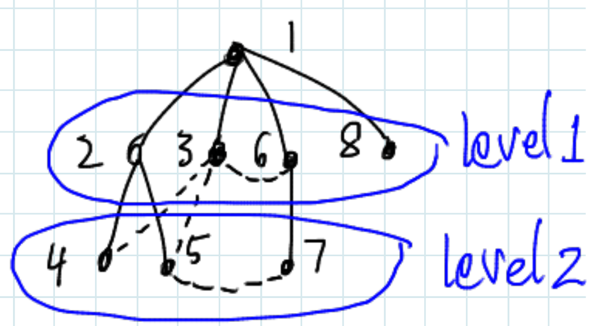


Cautious search: check everything one edge away, then two ...

order in which vertices are discovered

$$1, \underbrace{2,3,6,8}_{\text{1's neighbours}}, \underbrace{4,5}_{\text{2's}}, \underbrace{7}_{\text{6's}}$$

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 → discovered

Explore (v)
for each neighbour u of v
- if mark(u) = undiscovered
  mark(u) $\leftarrow$ discovered   parent (u) $\leftarrow$ v
  level (u) $\leftarrow$ level(v)+1
  add u to Queue
end

BFS

initialize: mark all vertices undiscovered
pick initial vertex v_0 parent(v_0) $\leftarrow \emptyset$ level(v_0) = 0
add v_0 to Queue; mark(v_0) $\leftarrow$ discovered
while Queue not empty
  $v \leftarrow$ remove from Queue
  Explore(v)
end

Also useful to store parent and level (see previous example) See blue additions above.

BFS takes $O(n+m)$ time. — we explore each vertex once and check all incident edges.

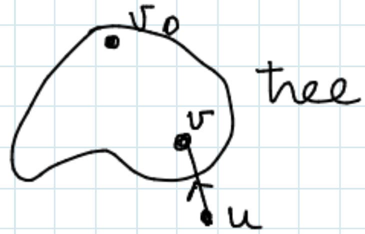
time is $O\left(n + \sum_v \deg(v)\right) = O(n+m)$.

Note: $\sum_v \deg(v) = 2m$

because we count each edge twice.

Properties of BFS

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



- u is connected to v_0 iff BFS from v_0 reaches u. Stronger:
- <u>Lemma</u>. the shortest path from v_0 to u has length (# edges) k iff BFS from v_0 puts u in level k.

<u>Proof</u> by induction with basis $k=0$

$\Leftarrow$ Suppose u in level k. Then parent$(u) = v$ is in level $k-1$. So shortest path v_0 to v has length $k-1$ by induction. There is a path v_0 to u of length k. Is it shortest? Yes, otherwise (by induction) u would be in a level $< k$.

$\Rightarrow$ suppose shortest path is $v_0, v_1, \ldots, v_k = u$ then $v_0 \cdots v_{k-1}$ is a shortest path of length $k-1$. So v_{k-1} goes in level $k-1$. Then u (a neighbour of v_{k-1}) goes in level $\leq k$. Could u go in level $< k$? No, otherwise (by ind.) there would be a shorter path to u.

Consequences:

1. BFS from v_0 finds the connected component of v_0.

   EX. Enhance BFS to find all connected components in time $O(n+m)$

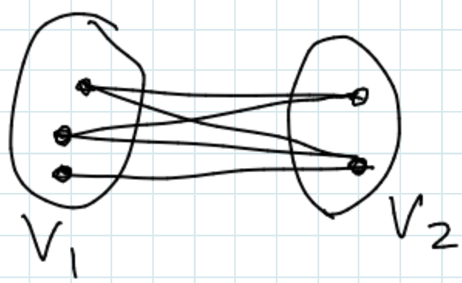
2. BFS finds shortest paths (# edges) from v_0.

EX. Use BFS to find if a connected graph has a cycle.

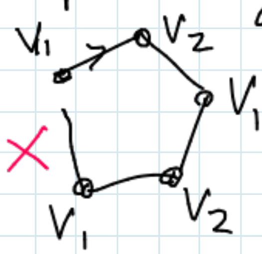
EX. prove that if $(u,v) \in E$ then $level(u)$, $level(v)$ differ by 0 or 1.

BFS to test bipartiteness

G is <u>bipartite</u> if V can be partitioned into $V_1 \cup V_2$ $(V_1 \cap V_2 = \emptyset)$ s.t. every edge has one end in V_1 and one end in V_2



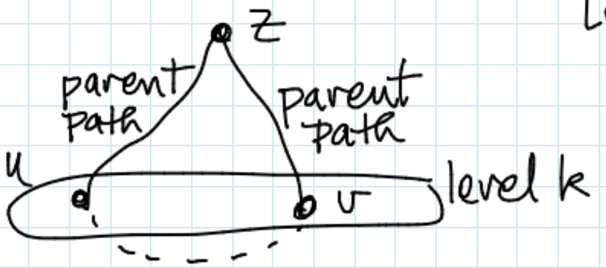
Note that a bipartite graph cannot have an odd cycle.



Run BFS. V_1 = odd levels V_2 = even levels.
Test if this works (check edges)
- if YES $\longrightarrow$ G is bipartite
- if NO then there is an edge (u,v) with u,v both in V_i ($i=1$ or 2)

By Ex. level(u) and level(v) differ by 1 or 0.
If 1, then one in V_1, one in V_2.
So u,v are in same level, say k.

Let z = least common ancestor of u,v.



Cycle formed by path(u,z) path(z,v) (u,v)
has length $2t+1$ — odd
Then G is not bipartite.
This proves:

<u>Lemma</u> G is bipartite iff it has no odd cycle.

the proof is via an algorithm the finds a bipartition OR an odd cycle.