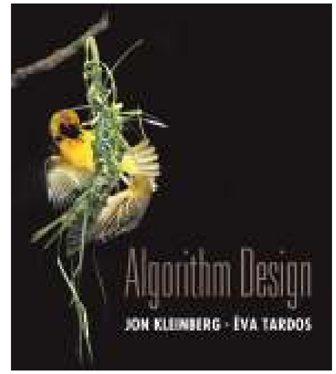
CSE 417: Algorithms and Computational Complexity

Winter 2012

Graphs and Graph Algorithms

Based on slides by Larry Ruzzo

1



Graphs

Reading: 3.1-3.6

2

Goals

Graphs: Definitions, examples, utility, terminology, representation

Traversal: Breadth- & Depth-first search

Three Algorithms:

- Connected Components
- Bipartiteness
- Topological sort

3

3.1 Basic Definitions and Applications

Graphs: Objects & Relationships

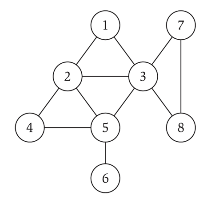
An extremely important formalism for representing (binary) relationships

Exam Scheduling:
- Classes
- Two are related if they have students in common

Traveling Salesperson Problem:
- Cities
- Two are related if can travel *directly* between them

Undirected Graphs

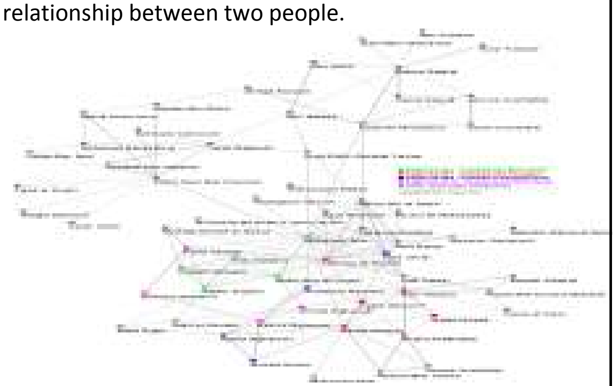
Undirected graph. G = (V, E)
- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

Social Network

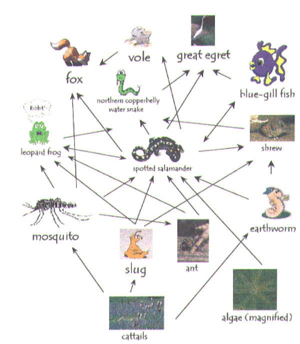
Node: people.
Edge: relationship between two people.



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

Vertices vs # Edges

Let G be an undirected graph with n vertices and m edges. How are n and m related?

Since

every edge connects two different vertices (no loops), and no two edges connect the same two vertices (no multi-edges),

It must be true that:

$0 \leq m \leq n(n-1)/2 = O(n^2)$

9

More Cool Graph Lingo

A graph is called *sparse* if $m << n^2$, otherwise it is *dense*

Boundary is somewhat fuzzy; O(n) edges is certainly sparse, $\Omega(n^2)$ edges is dense.

Sparse graphs are common in practice

E.g., all planar graphs are sparse ($m \leq 3n-6$, for $n \geq 3$)

Q: which is a better run time, O(n+m) or $O(n^2)$?

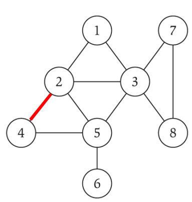
A: $O(n+m) = O(n^2)$, but n+m usually way better!

10

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with A_{uv} = 1 if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



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

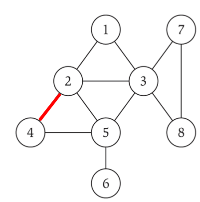
11

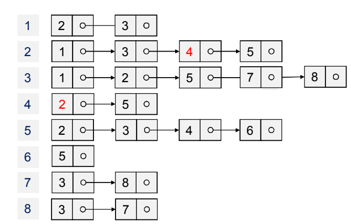
Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(deg(u)) time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree = number of neighbors of u





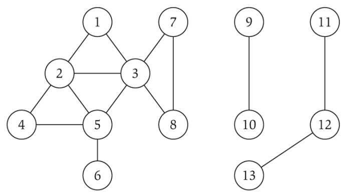
12

Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

Def. A path is simple if all nodes are distinct.

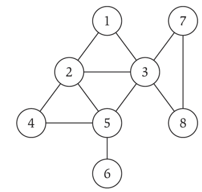
Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



13

Cycles

Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.



cycle C = 1-2-4-5-3-1

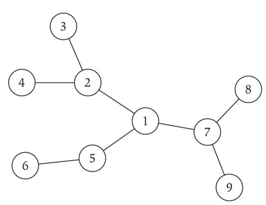
14

Trees

Def. 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.

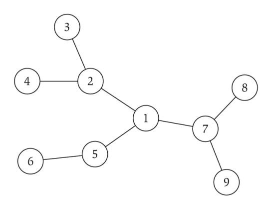


15

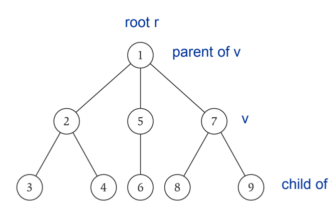
Rooted Trees

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

Importance. Models hierarchical structure.



a tree



the same tree, rooted at 1

16

3.2 Graph Traversal

Graph Traversal

Learn the basic structure of a graph

"Walk," *via edges*, from a fixed starting vertex *s* to all vertices reachable from *s*

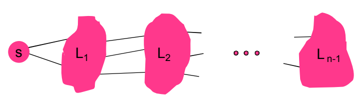
Being *orderly* helps. Two common ways:

- Breadth-First Search
- Depth-First Search

Breadth-First Search

Idea: Explore from s in all possible directions, layer by layer.

BFS algorithm.

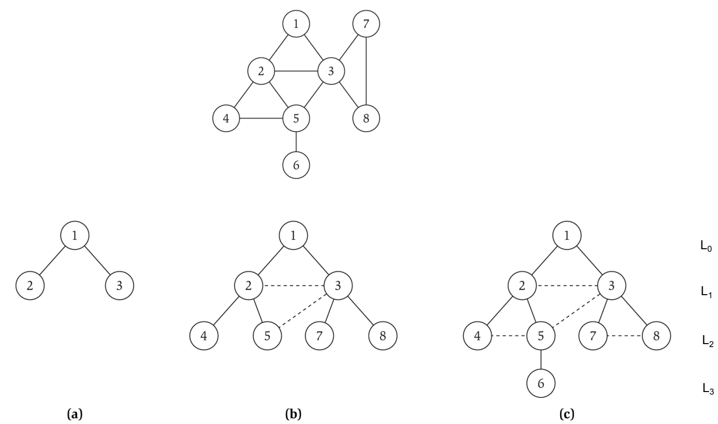


$L_0 = \{ s \}$.
L_1 = all neighbors of L_0.
L_2 = all nodes not in L_0 or L_1, and having an edge to a node in L_1.
L_{i+1} = all nodes not in earlier layers, and having an edge to a node in L_i.

Theorem. For each i, L_i consists of all nodes at distance (i.e., min path length) exactly i from s.

Cor: There is a path from s to t iff t appears in some layer.

Breadth First Search: Example



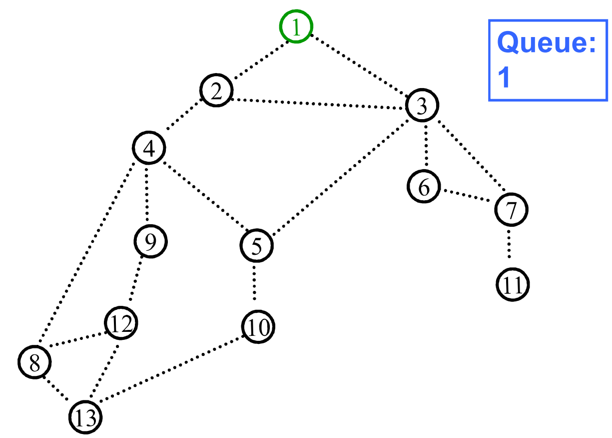
BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

```
BFS(s)
   Mark s "discovered"
   queue = {s}
   while queue not empty
      u = remove_first(queue)
      for each edge {u,x}
         if (x is undiscovered)
            mark x discovered
            append x on queue
      mark u fully explored
```

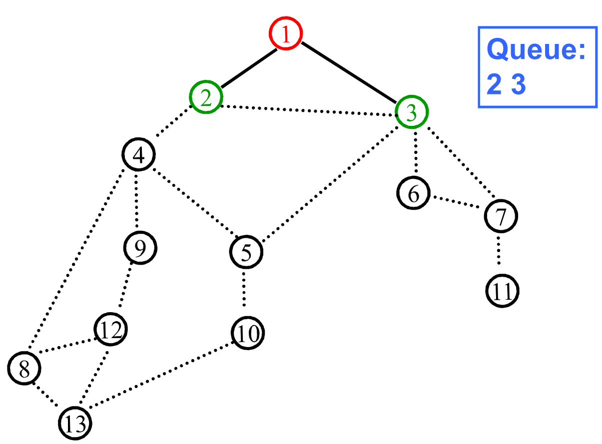
21

BFS(v)



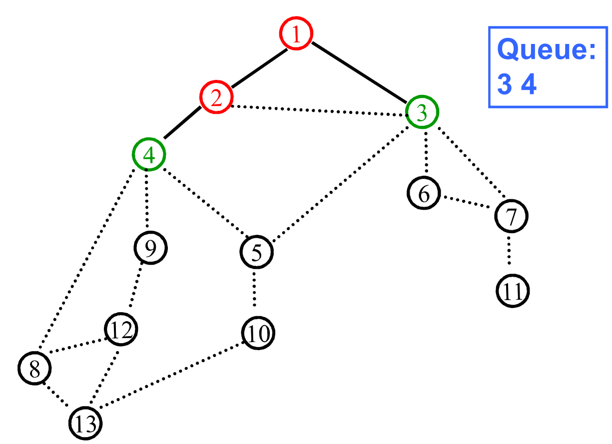
22

BFS(v)



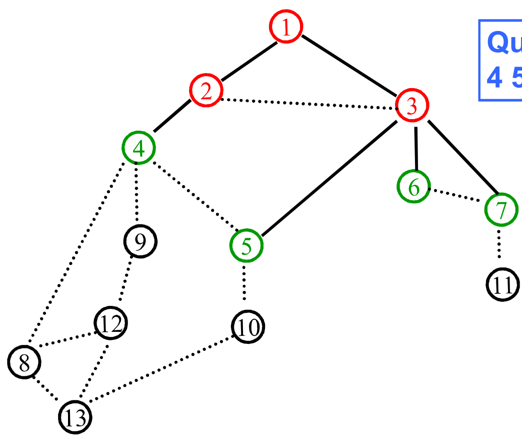
23

BFS(v)



24

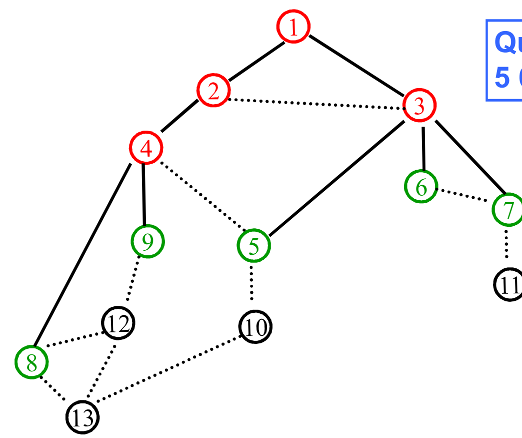
BFS(v)



Queue:
4 5 6 7

25

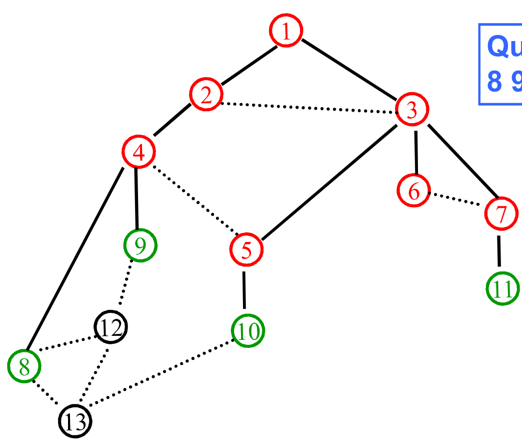
BFS(v)



Queue:
5 6 7 8 9

26

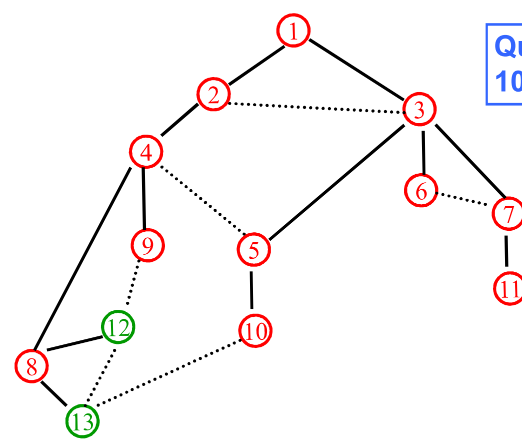
BFS(v)



Queue:
8 9 10 11

27

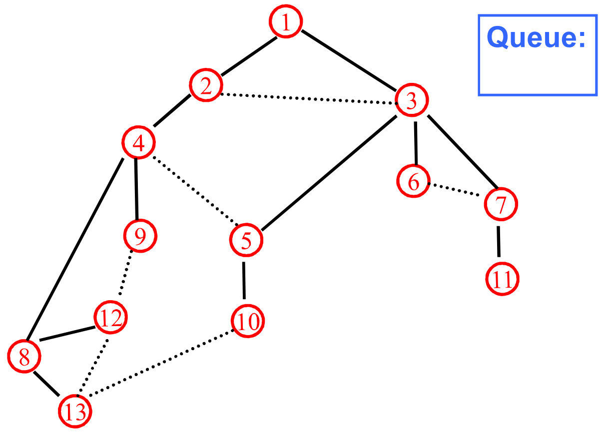
BFS(v)



Queue:
10 11 12 13

28

BFS(v)



Breadth First Search: Analysis

Theorem. The above implementation of BFS runs in O(??) time if the graph is given by its adjacency representation.

Pf. Easy to prove $O(n^2)$ running time:

- After a node is removed from the queue, it never appears in the queue again : while loop runs $\leq n$ times
- when we consider node u, there are $\leq n$ incident edges (u, v), and we spend O(1) processing each edge

Breadth First Search: Analysis

Actually runs in O(m + n) time:

- when we consider node u, there are deg(u) incident edges (u, v)
- total time processing edges is $\Sigma_{u \in V} \deg(u) = 2m$

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)

Homework Rules

Homework 2 is out, due next Wednesday in class.

When asked to describe an algorithm, you should give:

1. An English description of the algorithm idea
2. A pseudocode if the English description is not sufficient to communicate the fundamental details
3. (Optional) An example to illustrate the idea
4. A clear correctness proof if the proof is not a part of the algorithm description
5. A clear analysis of the running

Homework Rules

When asked to prove a statement

1. Make sure all your variables are defined
2. Never write an argument you are not convinced in because this may damage your brain
3. If the proof is long, explain the proof idea before explaining the details

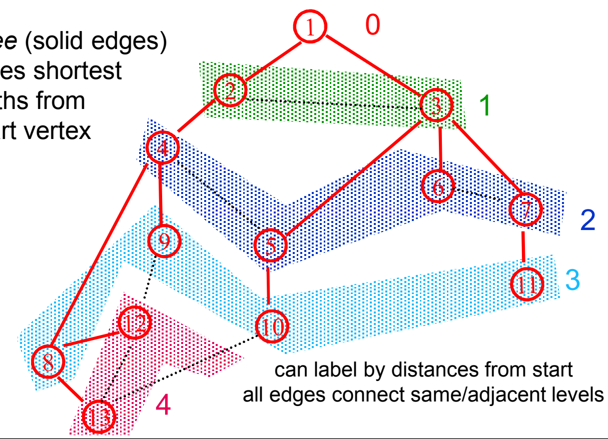
Format: Submit each problem on a SEPARATE sheet(s) of paper with your name and the problem number. Your homework will be disregarded if it is not in this format.

Due date rule: Late homeworks are not accepted.

33

BFS Application: Shortest Paths

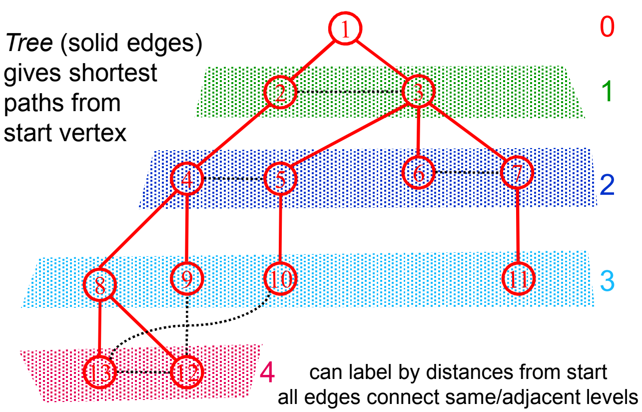
Tree (solid edges) gives shortest paths from start vertex



34

BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex



35

Why fuss about trees?

- Trees are simpler than graphs
- So, this is often a good way to approach a graph problem: find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure
- E.g., BFS finds a tree s.t. level-jumps are minimized

DFS (next) finds a different tree, but it also has an interesting structure…

36

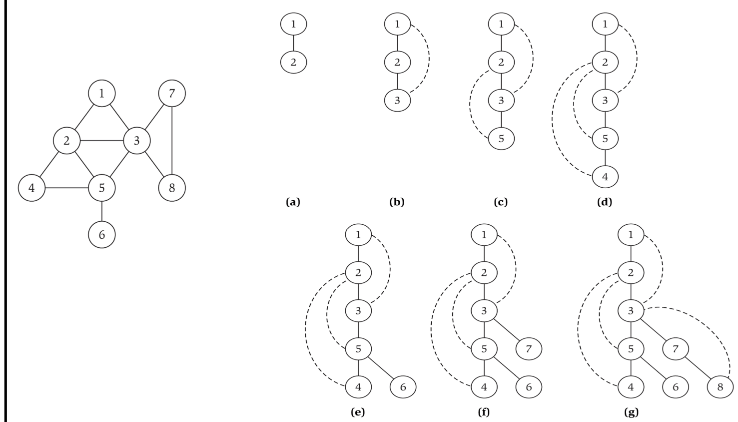
Depth-First Search

Idea: Follow the first path you find as far as you can go

Back up to last unexplored edge when you reach a dead end, then go as far you can

Naturally implemented using a stack

Depth First Search: Example



DFS(s) Implementation

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u,v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```


Depth First Search: Analysis

Theorem. The above implementation of BFS runs in O(m + n) time if the graph is given by its adjacency representation.

Pf. Similar ideas to BFS analysis

BFS vs DFS

Similarities:

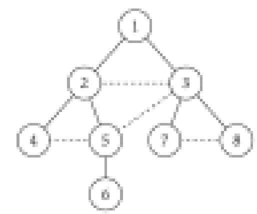
- Both visit x if and only if there is a path in G from v to x.
- Edges into then-undiscovered vertices define a **tree**

Differences:

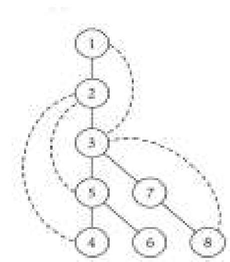
- In the BFS tree, levels reflect minimum distance from the root; not the case for DFS
- In BFS, all non-tree edges join vertices on the same or adjacent levels while in DFS, all non-tree edges join a vertex and one of its descendants/ancestors in the DFS tree

41

BFS vs DFS



BFS tree

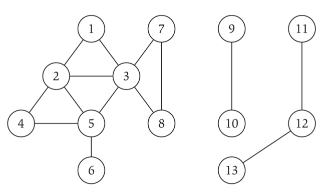


DFS tree

42

Graph Search Application: Connected Component

Connected component. Find all nodes reachable from s.



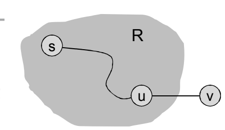
Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

43

Graph Search Application: 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 $u \in R$ and $v \notin R$
  Add v to R
Endwhile



it's safe to add v

Theorem. Upon termination, R is the connected component containing s.

BFS = explore in order of distance from s.

DFS = explore in a different way.

44

3.4 Testing Bipartiteness

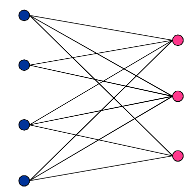
Bipartite Graphs

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

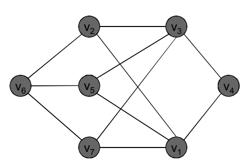
Testing Bipartiteness

Given a graph G, is it bipartite?

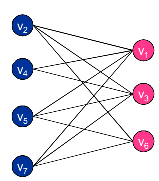
Many graph problems become:

- 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

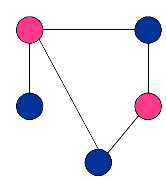


another drawing of G

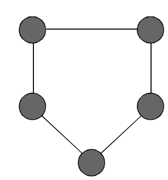
An Obstruction to Bipartiteness

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)



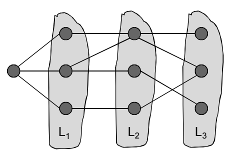
not bipartite
(not 2-colorable)

Bipartite Graphs

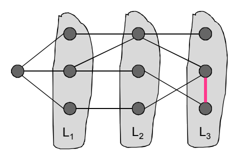
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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).



Case (i)



Case (ii)

49

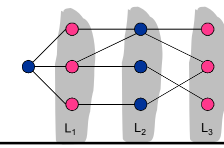
Bipartite Graphs

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

Pf.

- Suppose no edge joins two nodes in the same layer.
- By the properties of a BFS tree, this implies all edges join nodes on adjacent levels.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.
- All edges have differently colored endpoints.

Case (i)



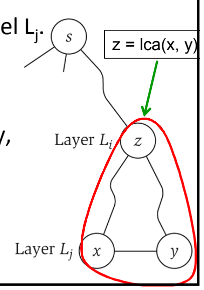
50

Bipartite Graphs

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

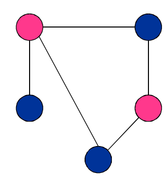
Pf.

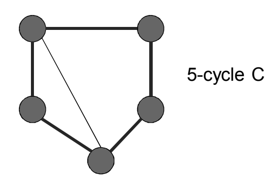
- Suppose (x, y) is an edge with x, y in same level L_j.
- Let z = lca(x, y) = lowest common ancestor.
- Let L_i be level containing z.
- Consider the cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from y to z}} + \underbrace{(j-i)}_{\text{path from z to x}}$, 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)

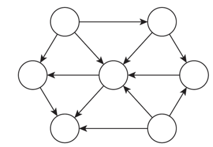
52

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

Graph search. BFS extends naturally to directed graphs.

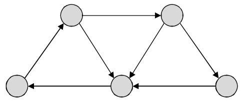
Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

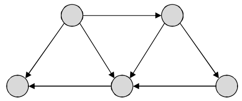
Strong Connectivity

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.



strongly connected



not strongly connected

Strong Connectivity

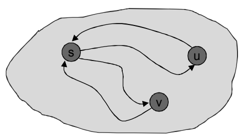
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf.

⇒ Follows from definition.

⇐ Let u and v be arbitrary nodes in G

1. Path from u to v: concatenate u-s path with s-v path.
2. Path from v to u: concatenate v-s path with s-u path.



Strong Connectivity: Algorithm

Theorem. We can determine if G is strongly connected in O(m + n) time.

Pf.

1. Pick any node s.
2. Run BFS from s in G.
3. Run BFS from s in G^{rev}. ← reverse orientation of every edge in G
4. Return true iff all nodes reached in both BFS executions.
5. Correctness follows immediately from previous lemma.

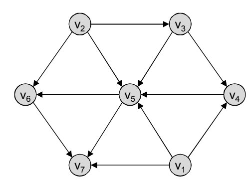
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

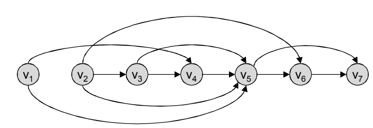
Def. An DAG is a directed graph that contains no directed cycles.

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

Edge (v_i, v_j) means task v_i must occur before v_j.

Applications

- Course prerequisites: course v_i must be taken before v_j
- Compilation: must compile module v_i before v_j
- Job Workflow: output of job v_i is part of input to job v_j
- Manufacturing or assembly: sand it before you paint it…
- Spreadsheet evaluation: cell v_j depends on v_i

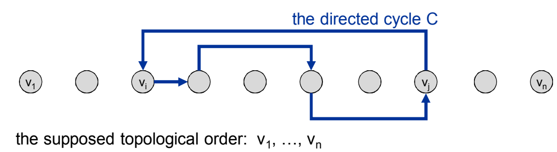
61

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, …, v_n and that G also has a directed cycle C.
- Let v_i be the lowest-indexed node in C, and let v_j be the node in C 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, …, v_n is a topological order, we must have j < i, a contradiction.



62

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?

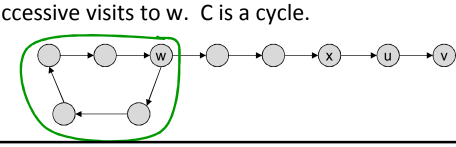
63

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.
- 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. Repeat same process for u
- 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.



64

Directed Acyclic Graphs

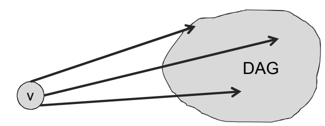
Lemma. If G is a DAG, then G has a topological ordering.

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.

Topological Sort Algorithm

```
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

Maintain the following:

count[w] = (remaining) number of incoming edges to node w

S = set of (remaining) nodes with no incoming edges

Initialization:

count[w] = 0 for all w

count[w]++ for all edges (v,w)

S = S ∪ {w} for all w with count[w]==0

} O(m + n)

Main loop:

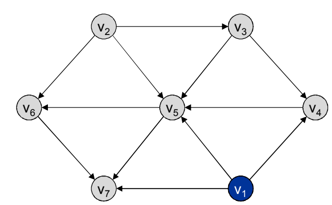
while S not empty

- remove some v from S
- make v next in topo order
- for all edges from v to some w
  - decrement count[w]
  - add w to S if count[w] hits 0

} O(1) per node, O(1) per edge

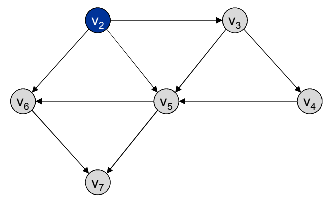
Time: O(m + n) (assuming edge-list representation of graph)

Topological Ordering Algorithm: Example



Topological order:

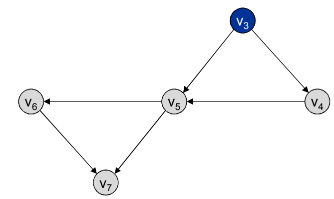
Topological Ordering Algorithm: Example



Topological order: v_1

69

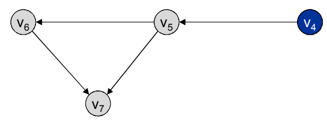
Topological Ordering Algorithm: Example



Topological order: v_1, v_2

70

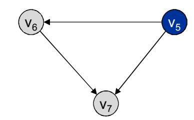
Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3

71

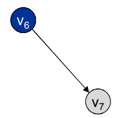
Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

72

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4, v_5

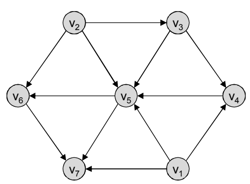
73

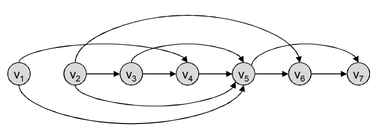
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

74

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

75