Graph

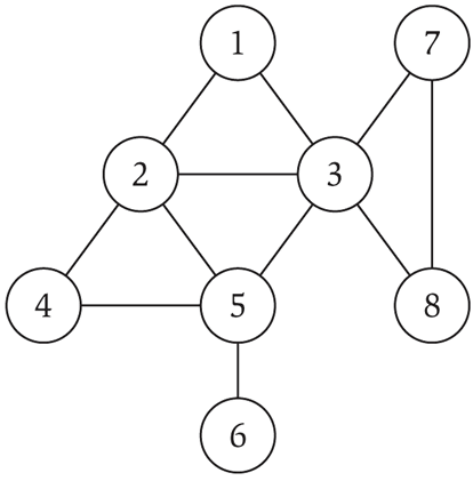
- Basic Definition and Applications
- Graph Traversal (BFS and DFS)
- Connected Componnents
- Testing Bipartiteness
- Connectivity in Directed Graphs
- DAGs and Topological Ordering
- Shortest Paths in a Graph
- Negative Cycles in a Graph

Basic Definitions and Applications

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

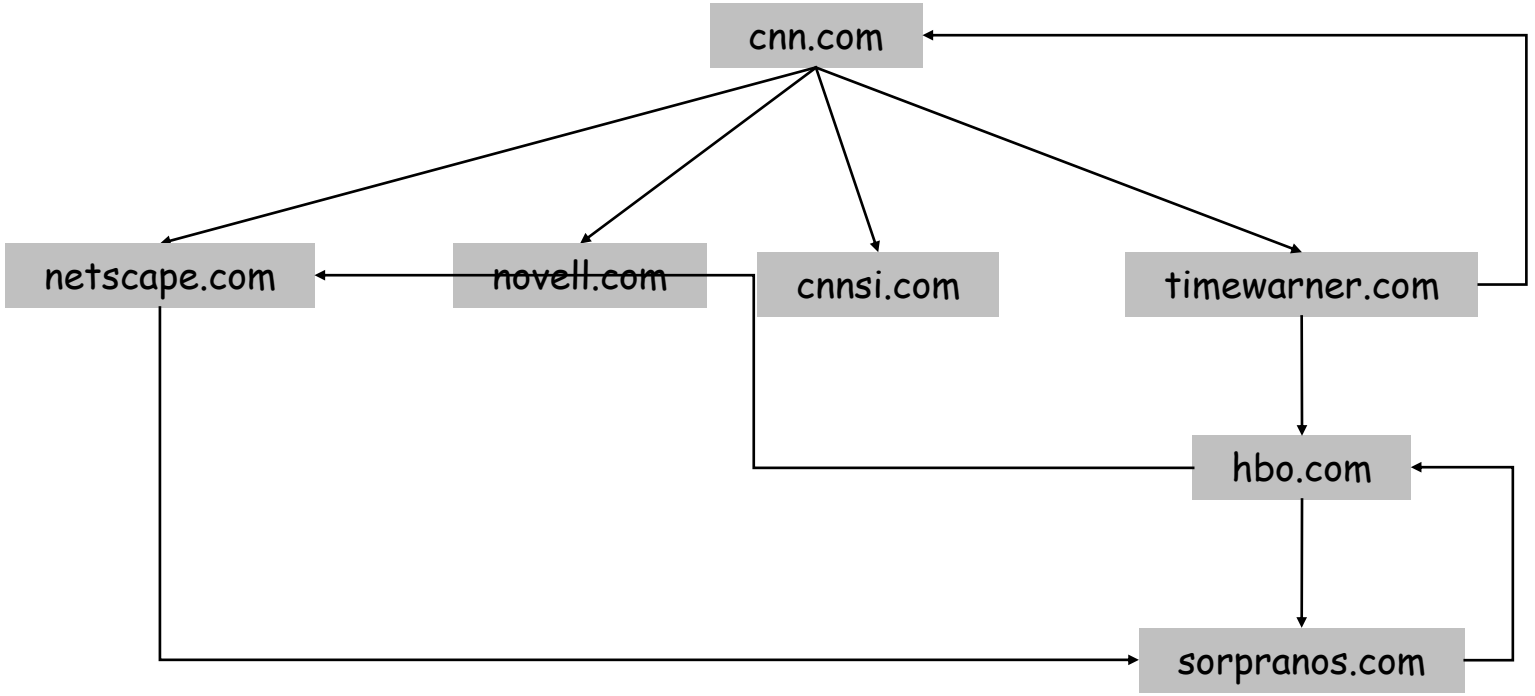
Some Graph Applications

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.

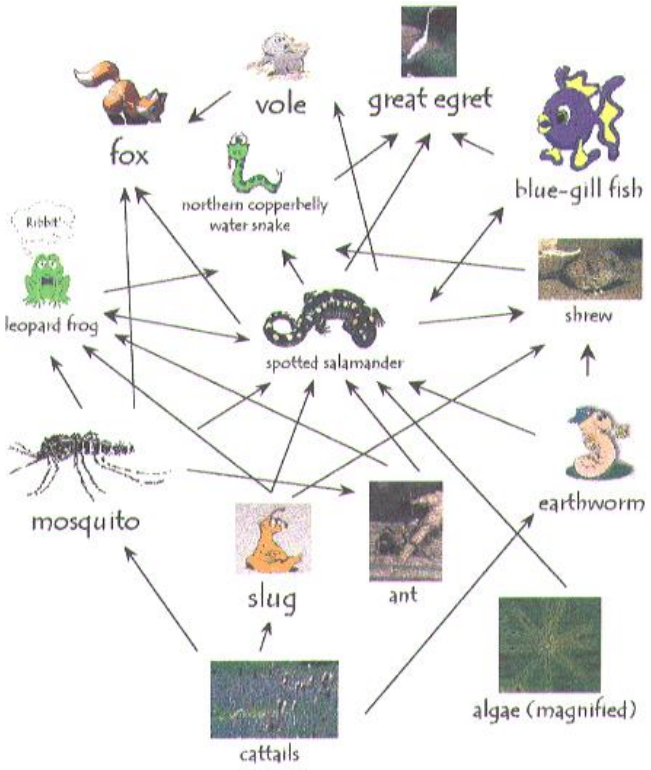
- Node: web page.
- Edge: hyperlink from one page to another.



Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

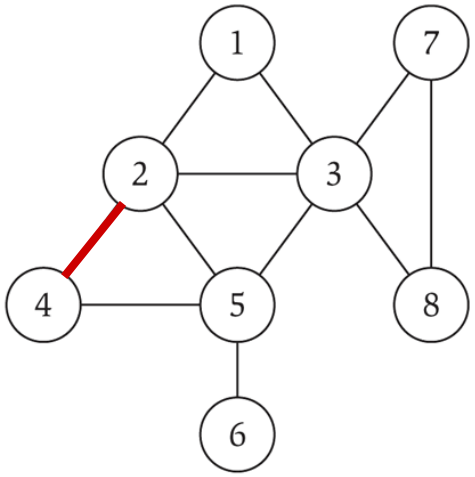


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

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.



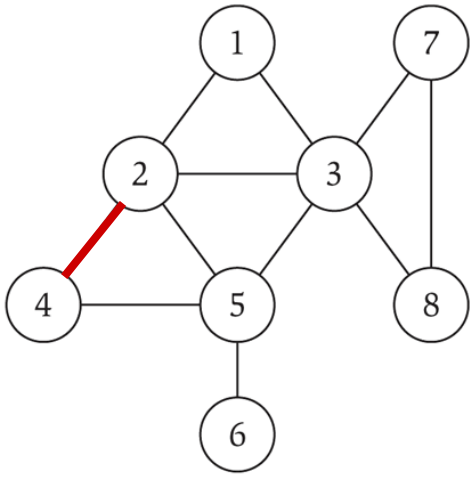
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

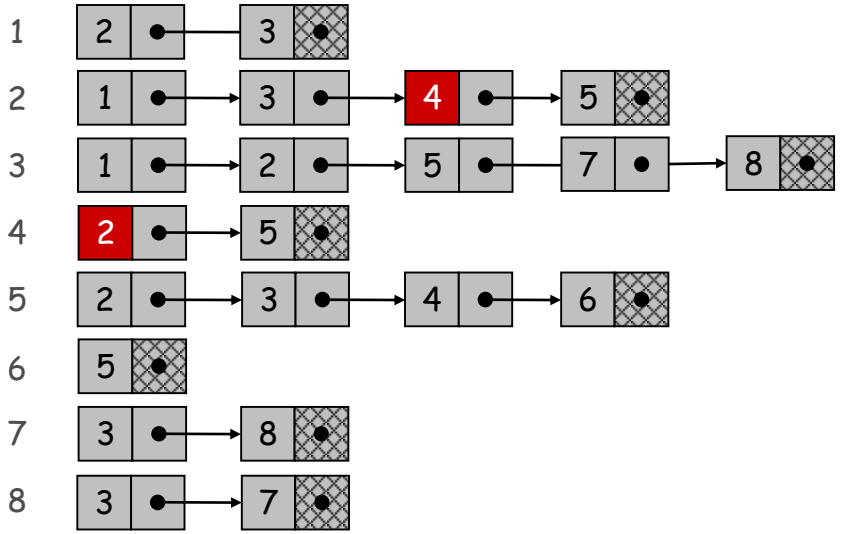
Graph Representation: Adjacency List

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



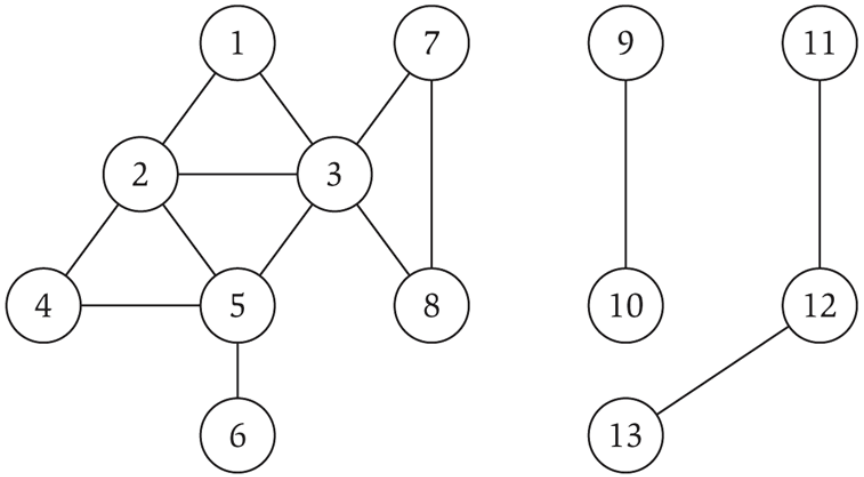


Paths and Connectivity

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.

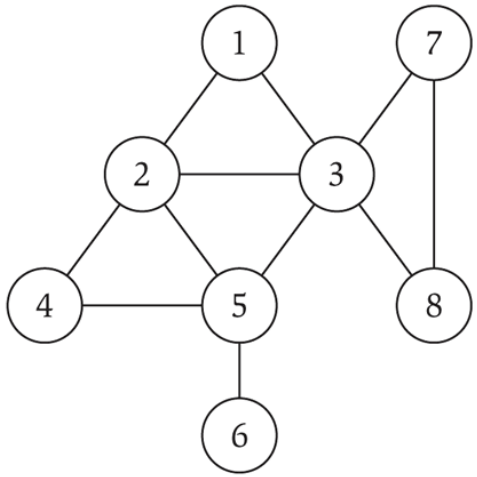
Def. A path is simple if all nodes are distinct.

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



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.



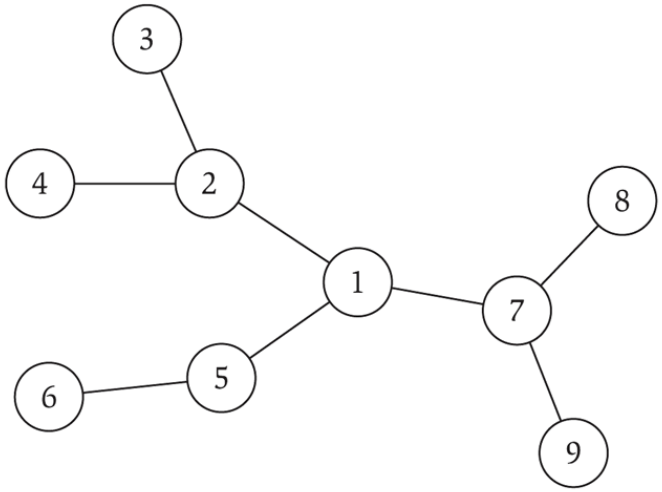
cycle C = 1-2-4-5-3-1

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.

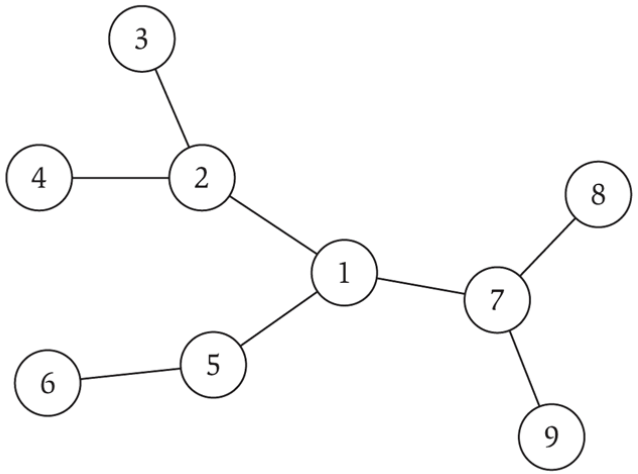
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



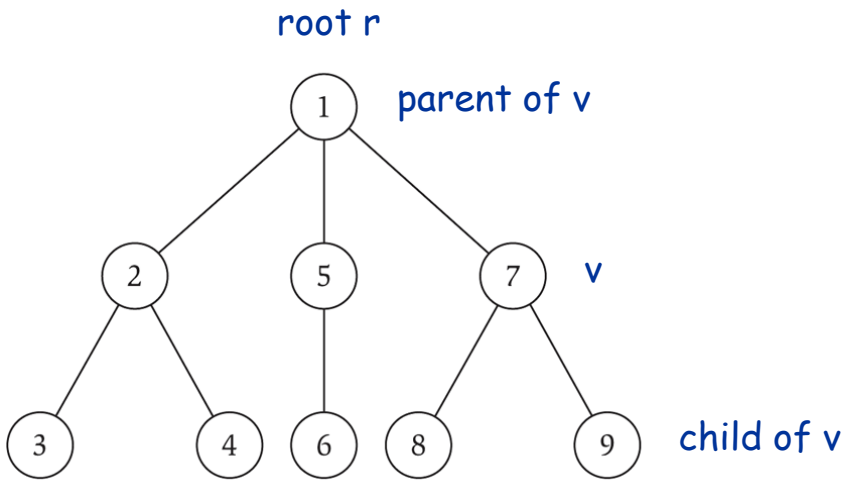
Rooted Trees

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

Graph Traversal

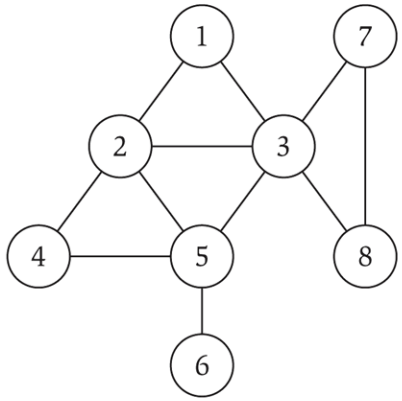
Connectivity

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.
- Erdos number.
- Fewest number of hops in a communication network.



Basic Graph Search Algorithm

Algorithm: SmartExplore(u): Initialize R = {u};
    Mark all vertices as unvisited; Mark u as visited;
    while R *is not empty* **do**
        Pick one vertex x in R, remove x from R
        foreach *vertex* y $\in$ Adj(x) **do** **if** y *is not visited* **then**
            Mark y as visited and add y to R
    return the set of all visited vertices.

- Runs in O(m + n) time.
- How to determine which vertex to pick in R

The Data Structure

Alternative 1: Queue

- First in first out (FIFO)
- Breadth First Search (BFS)
- Exploring distances

Alternative 2: Stack

- Last in first out (LIFO)
- Depth First Search (DFS)
- Exploring

Breadth First Search

Queue

A queue is a linked list with two operations

- Enqueue(Q, x): insert an element x at the rear of the queue
- Dequeue(Q): remove the front element of the queue.

Implementation:

- linked list with two pointers
- array with two pointers

Breadth First Search

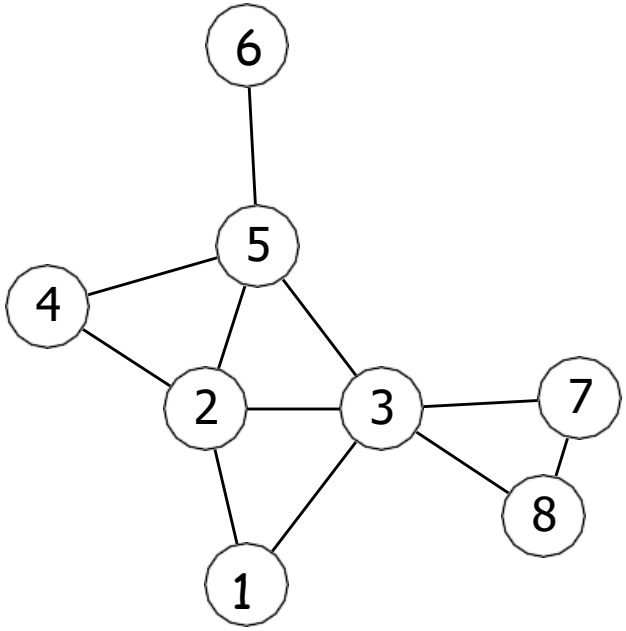
Algorithm: BFS(u):

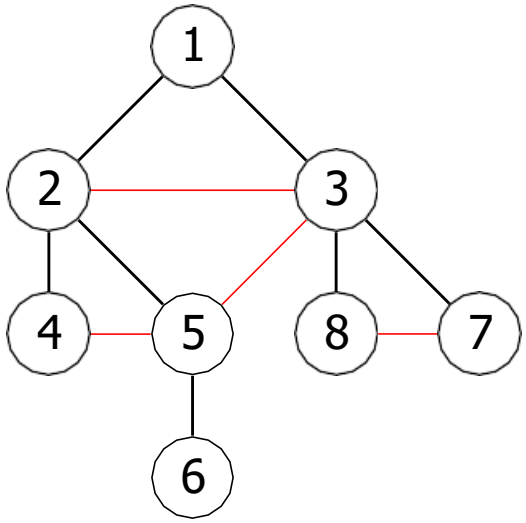
Initialize queue Q to be empty Mark all vertices as unvisited
Initialize search tree T to be empty
Mark u as visited and enqueue(Q, u)
while Q *is not empty* **do**
    x = dequeue(Q)
    foreach *vertex* y $\in$ Adj(x) **do** **if** y *is not visited* **then**
        add edge (x, y) to T
        Mark y as visited and enqueue(Q, y)

Proposition

BFS(u) runs in O(m + n) time

BFS: An Example





- BFS tree is the set of black edges.

Breadth First Search with Distances

Algorithm: BFS(u):

Initialize queue Q to be empty

Mark all vertices as unvisited

set dist(v) = ∞ for each v

Initialize search tree T to be empty

Mark u as visited and enqueue(Q, u)

dist(u) = 0

while Q *is not empty* **do**

    x = dequeue(Q);

    foreach *vertex* y $\in$ Adj(x) **do**

        if y *is not visited* **then**

            add edge (x, y) to T

            Mark y as visited and enqueue(Q, y)

            dist(y) = dist(x) + 1

Shortest Distance

Properties

- If dist(u) $<$ dist(v), then u is visited before v
- If e = (u, v) is an edge of G, then |dist(u) – dist(v)| ≤ 1

The shortest distance $\delta(u, v)$ between two vertices u and v in an unweighted graph G is the length of a shortest path (in terms of # of edges) from u to v.

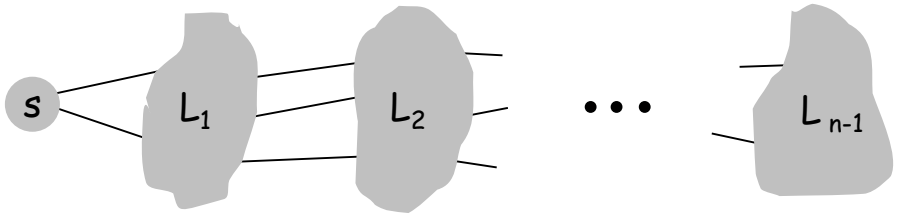
no path between u and v means $\delta(u, v) = \infty$

Proposition

Upon termination of BFS(u), for every vertex v, dist(v) = $\delta(u, v)$.

BFS Summary

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

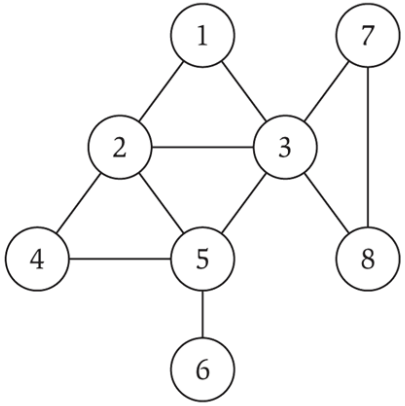


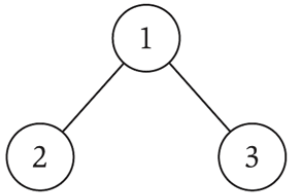
BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

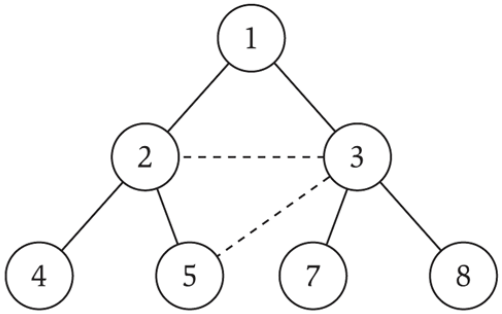
Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

BFS Summary

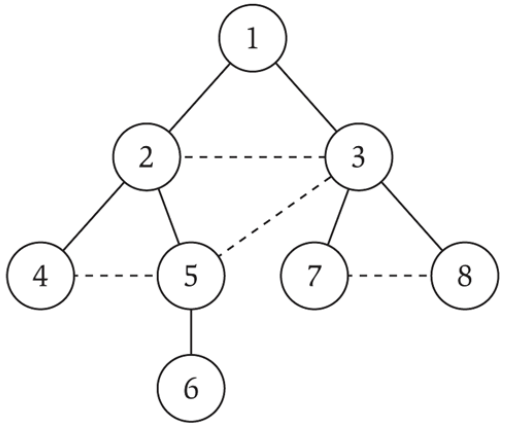




(a)



(b)



(c)

L_0

L_1

L_2

L_3

Depth First Search

A versatile graph exploration strategy.

Can be used to solve many nontrivial problems in linear time (O(m + n)).

- Finding cut-edges and cut-vertices of undirected graphs.
- Finding strongly connected components of directed graphs.
- Testing whether a graph is planar.

Basic Graph Search Algorithm with a stack.

Depth First Search

Stack

A stack is a linked list with two operations

Push(S, x): insert an element at the *front* of the stack.

Pop(S): remove the front element of the stack.

- Elements are processed in a last-in first-out (LIFO) order, different from the first-in first-out (FIFO) order for queues.
- Implementation: need to maintain only the pointer of the front of the stack.
- Useful to also have Peek(S): retrieve

Depth First Search

Algorithm: DFS(u):

```
Initialize stack S to be empty
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark u as visited and push(S, u)
while S is not empty do
    x = peek(S)
    foreach vertex y ∈ Adj(x) do if y is not visited
        then
            Mark y as visited and push(S, y)
    if there is no such y then
        pop(S)
```

A Recursive DFS

Algorithm: DFS(x):

```
Mark x as visited;
foreach vertex y ∈ Adj(x) do
    if y is not visited then
        add edge (x, y) to T;
        DFS(y);
```

Algorithm: DFS(G):

```
Mark all vertices as unvisited;
Set T to be empty;
while ∃ unvisited vertex u do
    DFS(u);
return T
```

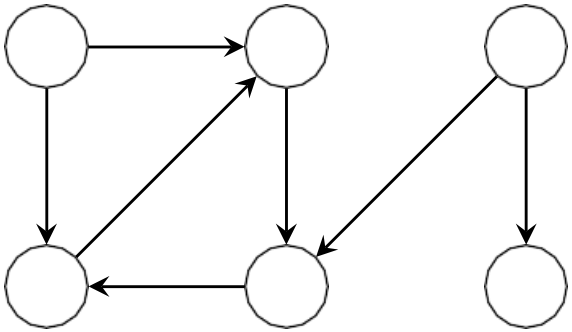
DFS Intuition

- exploring a maze
- from current vertex, move to

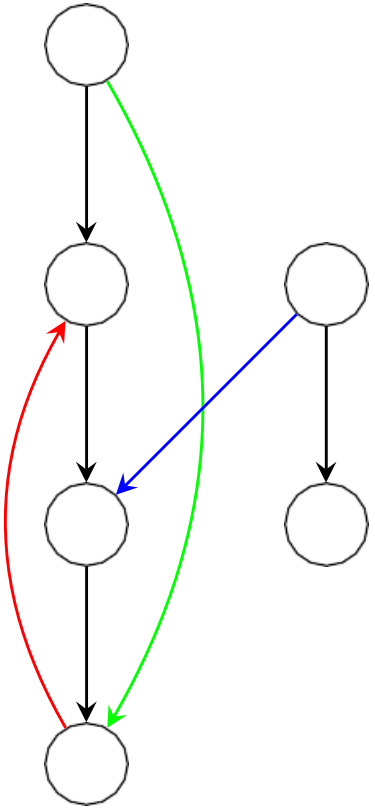
  another until you get stuck

- then backtrack till you find the first new possibility for exploration

DFS: An Example



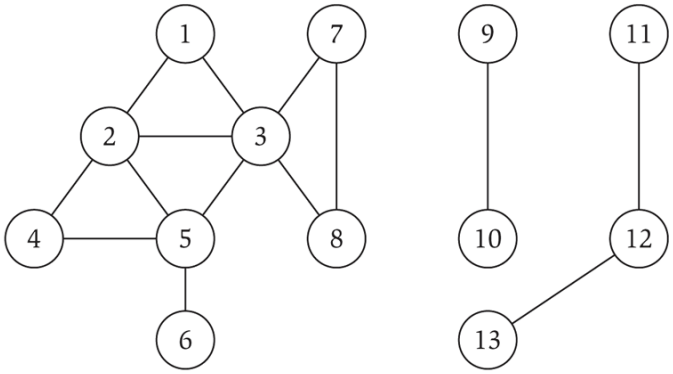
- tree edges
- back edges
- forward edges
- cross edges



Connected Component

Connected Component

Connected component. Find all nodes reachable from s.

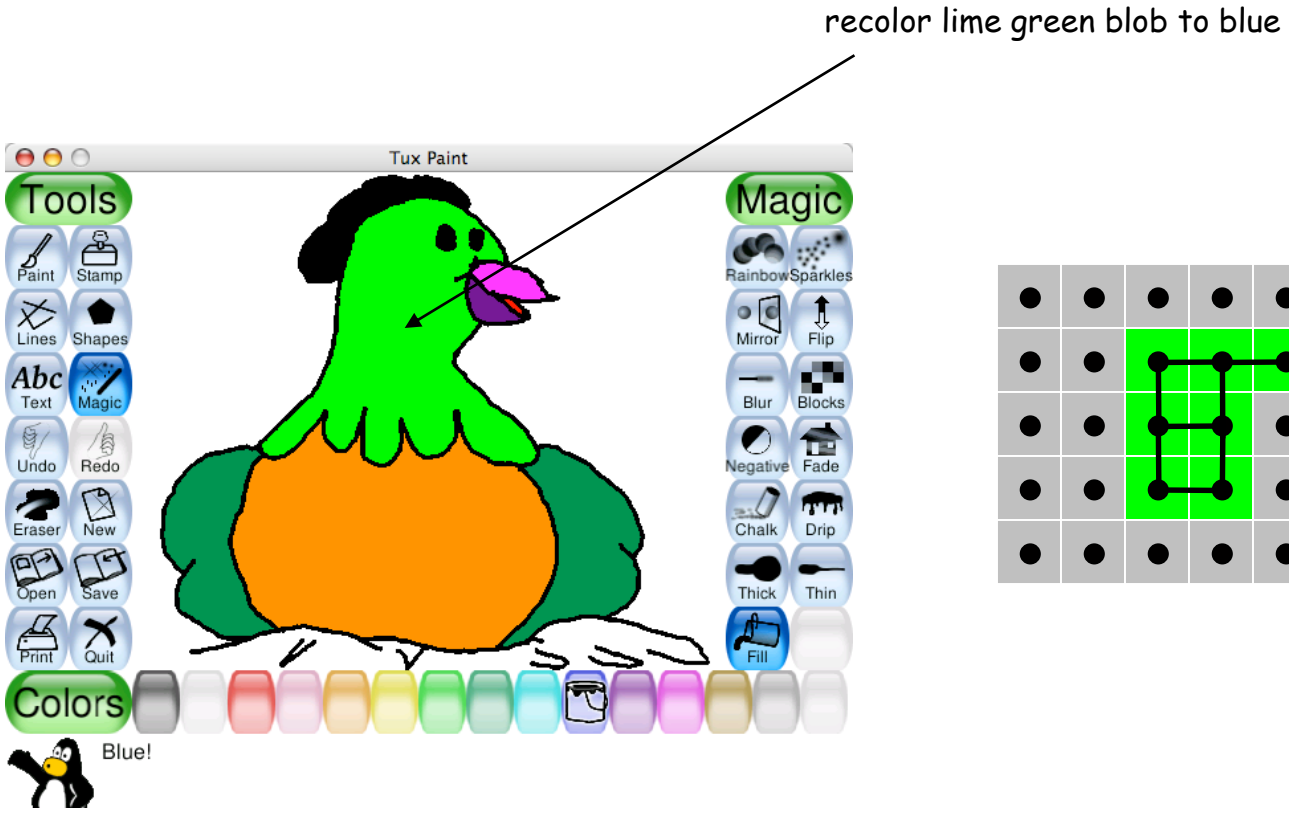


Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

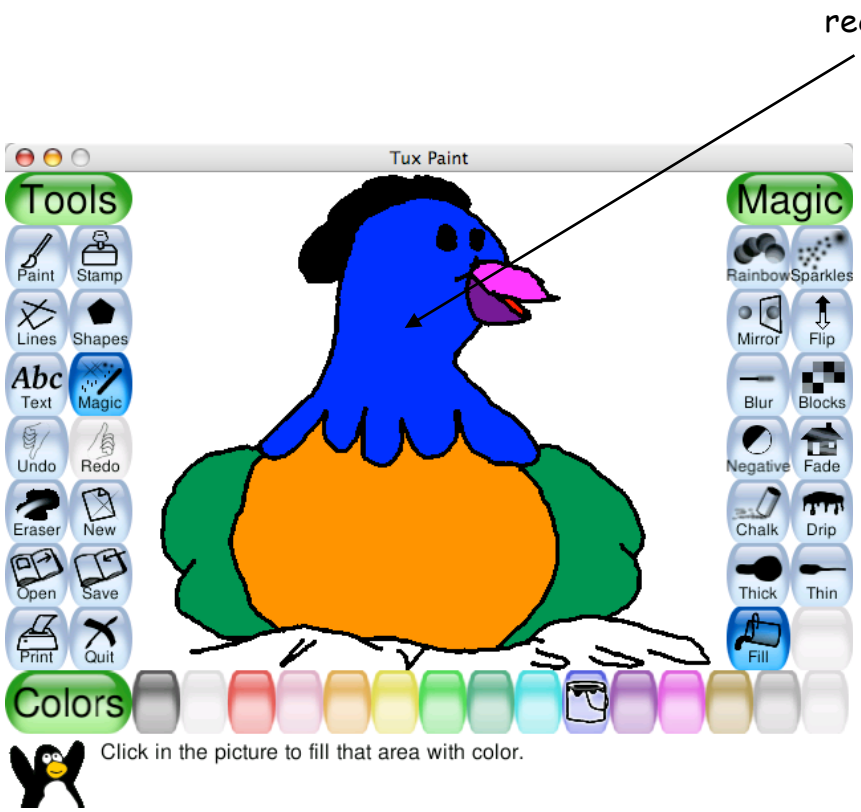
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

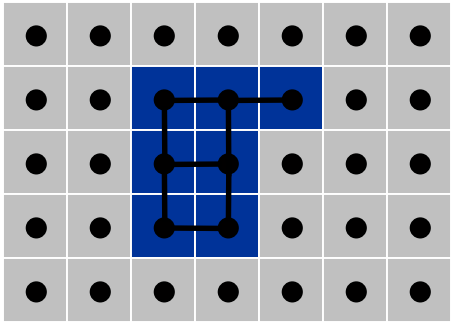


Flood Fill

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.

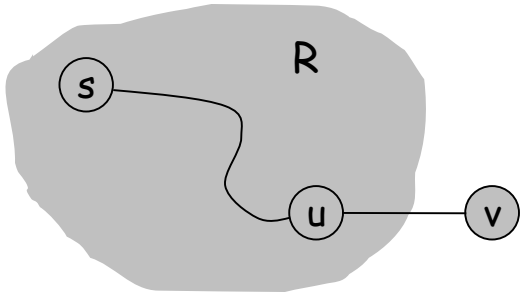




Connected Component

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.

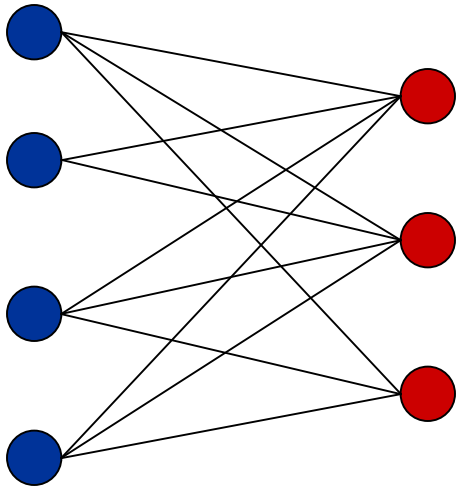
Testing Bipartiteness

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.

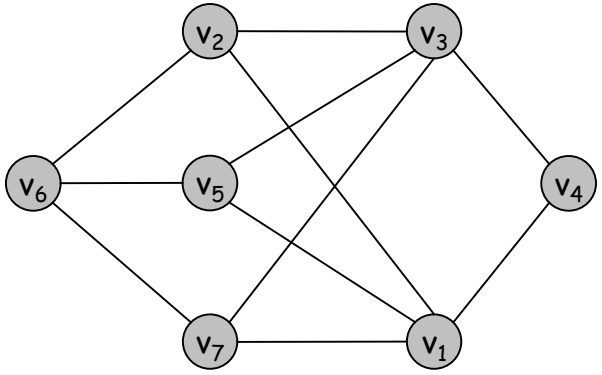


a bipartite graph

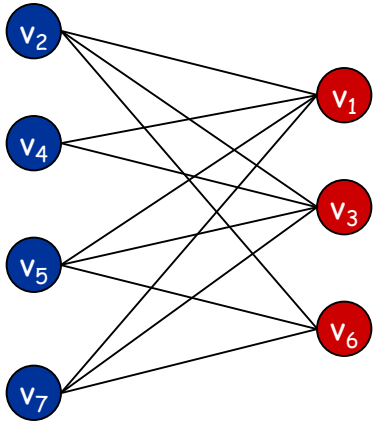
Testing Bipartiteness

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

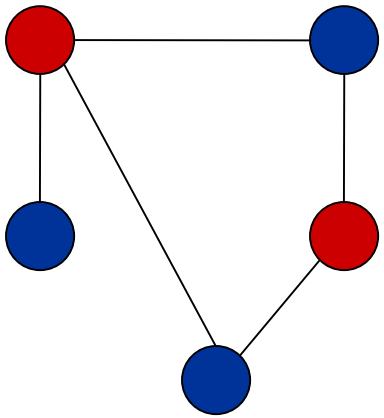


another drawing of G

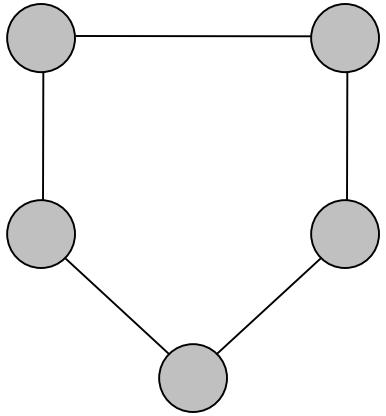
An Obstruction to Bipartiteness

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)



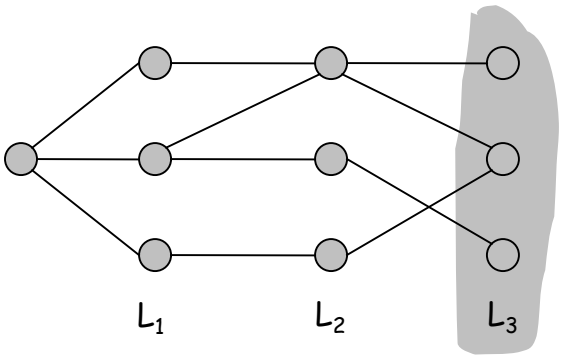
not bipartite
(not 2-colorable)

Bipartite Graphs

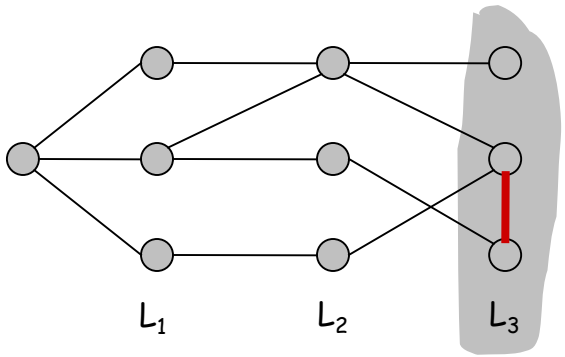
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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)

Bipartite Graphs

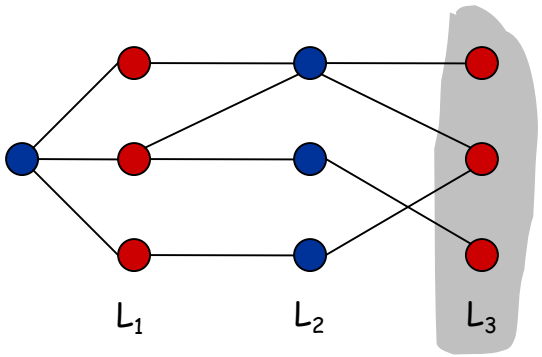
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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 adjacent layers.
- 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.

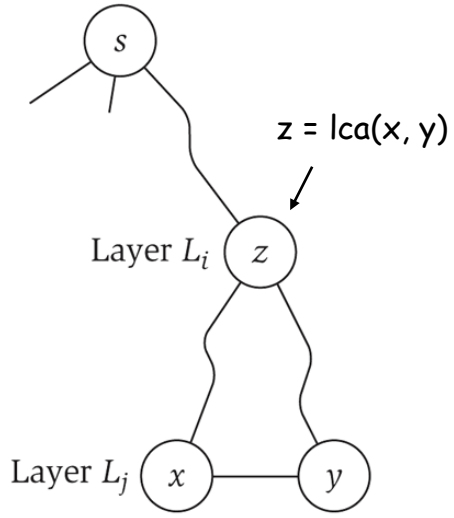
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(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)

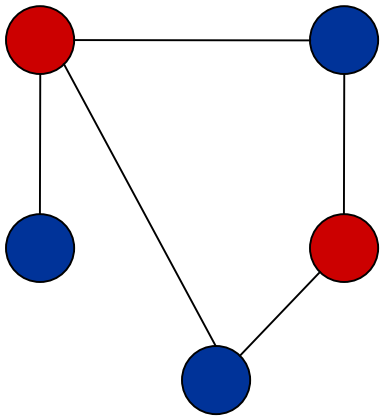
- Suppose (x, y) is an edge with x, y in same level L_j.
- Let z = lca(x, y) = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $1 + (j-i) + (j-i)$, which is odd. ▪

1: (x, y); $(j-i)$: path from y to z; $(j-i)$: path from z to x

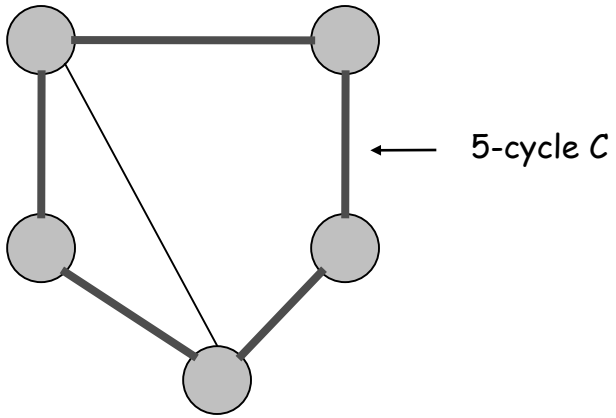


Obstruction to Bipartiteness

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



bipartite
(2-colorable)



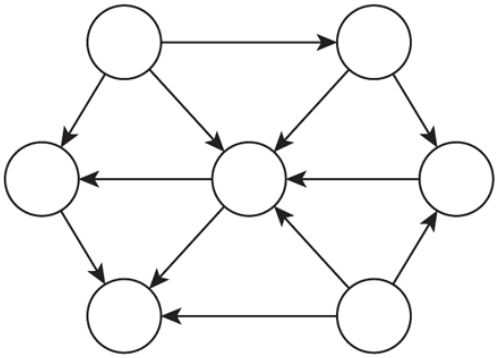
not bipartite
(not 2-colorable)

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.

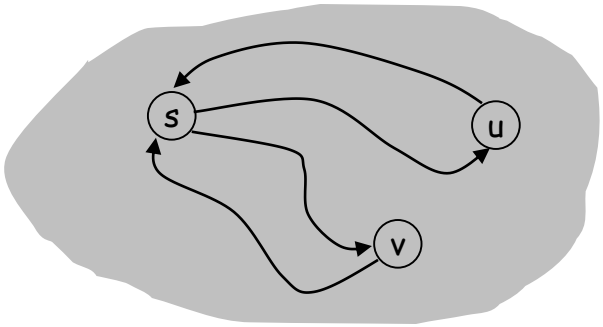
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.

Pf. $\Leftarrow$ 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

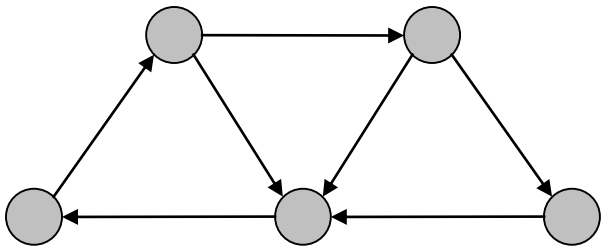


Strong Connectivity: Algorithm

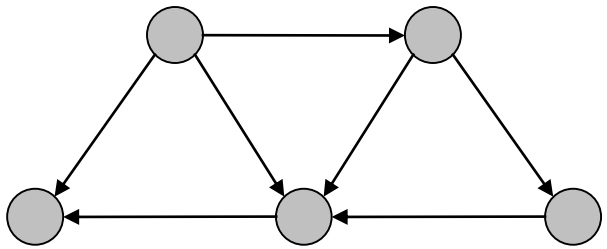
Theorem. Can determine if G is strongly connected in $O(m + n)$ time.

Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}. (reverse orientation of every edge in G)
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



strongly connected



not strongly connected

DAGs and Topological Ordering

Application: Topological Sorting

directed acyclic graph (DAG)

- Generalize trees to directed graphs, with much richer structure
- Can be used to encode *precedence relations* or *dependence* in a natural way

Question

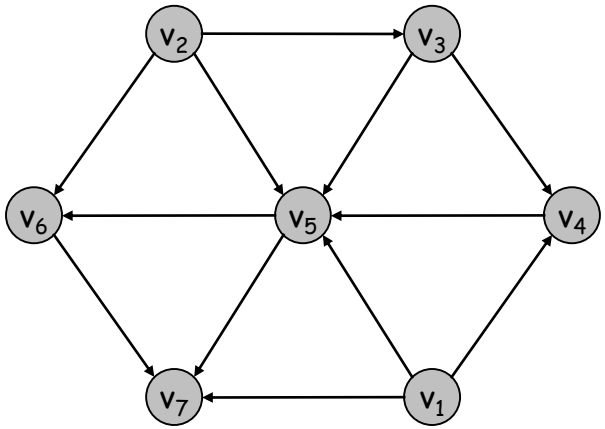
How to check if a given graph G is a DAG?

Directed Acyclic Graphs

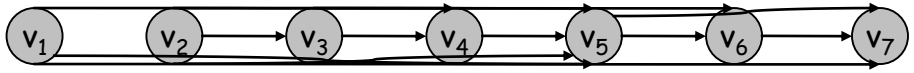
Def. An DAG is a directed graph that contains no directed cycles.

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.

How to check if a given graph G is a DAG

Lemma

Graph G is a DAG if and only if there are no back edges in its DFS tree.

Proof

$\Rightarrow$: If there is a back edge, then there is a cycle.

$\Leftarrow$: Assume there is a cycle C, let u be the first vertex discovered on C.

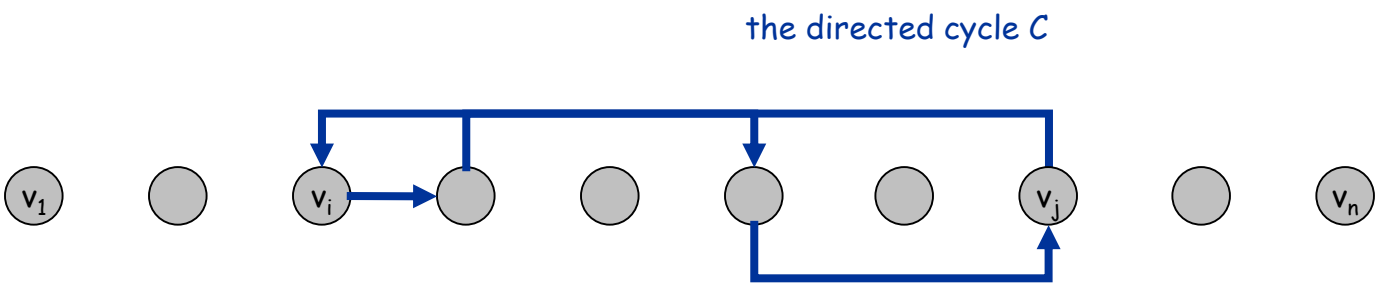
- Let (v, u) be the preceding edge in C. Then v is a descendant of u.
- (v, u) is a back edge.

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. ▪



Directed Acyclic Graphs

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

Q. Does every DAG have a topological ordering?

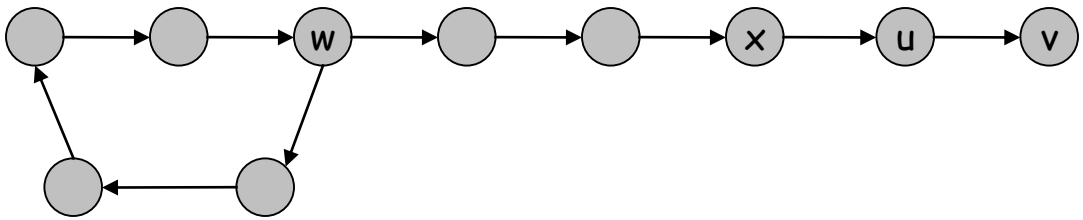
Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

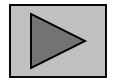
- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward 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

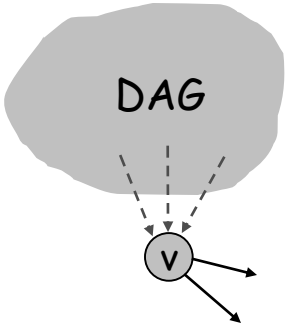
Lemma. If G is a DAG, then G has a topological ordering.

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



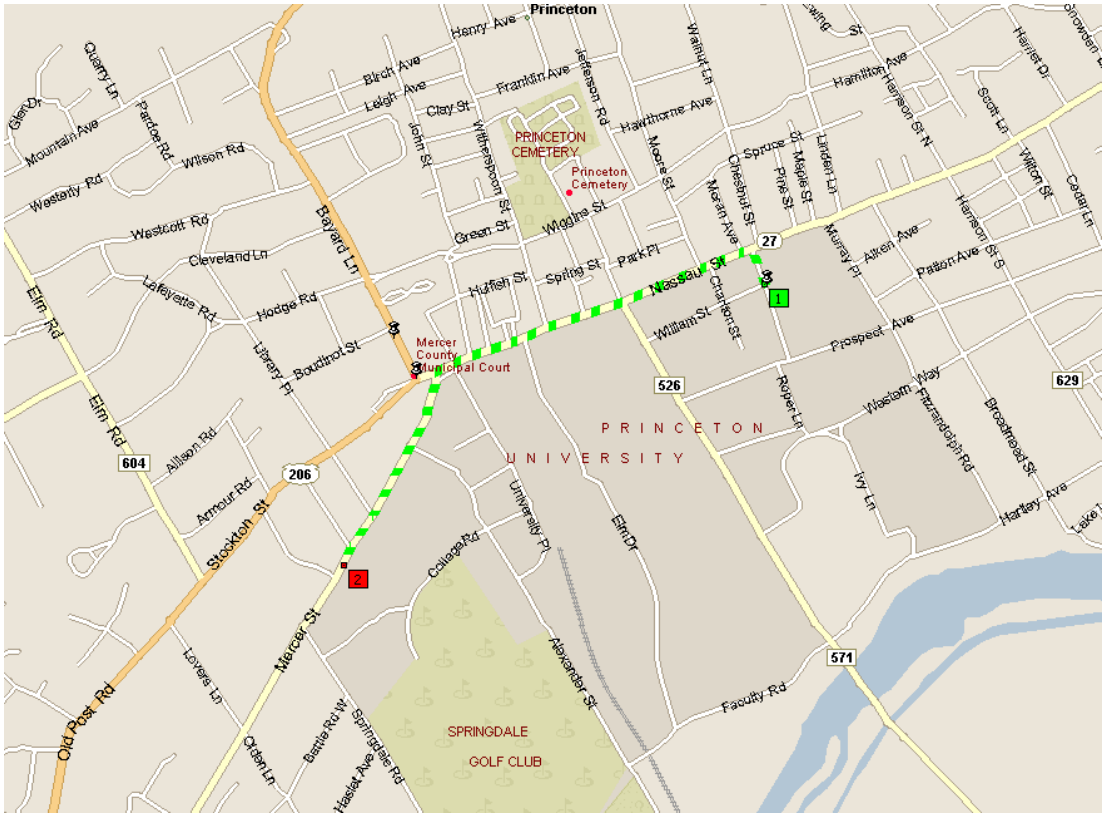
Topological Sorting Algorithm: Running Time

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 ▪

Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

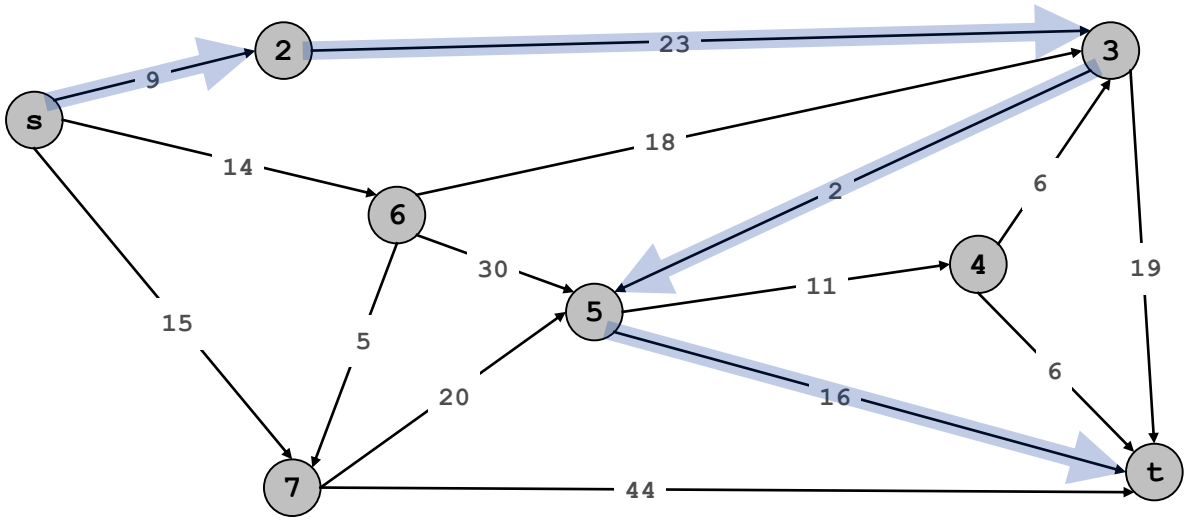
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



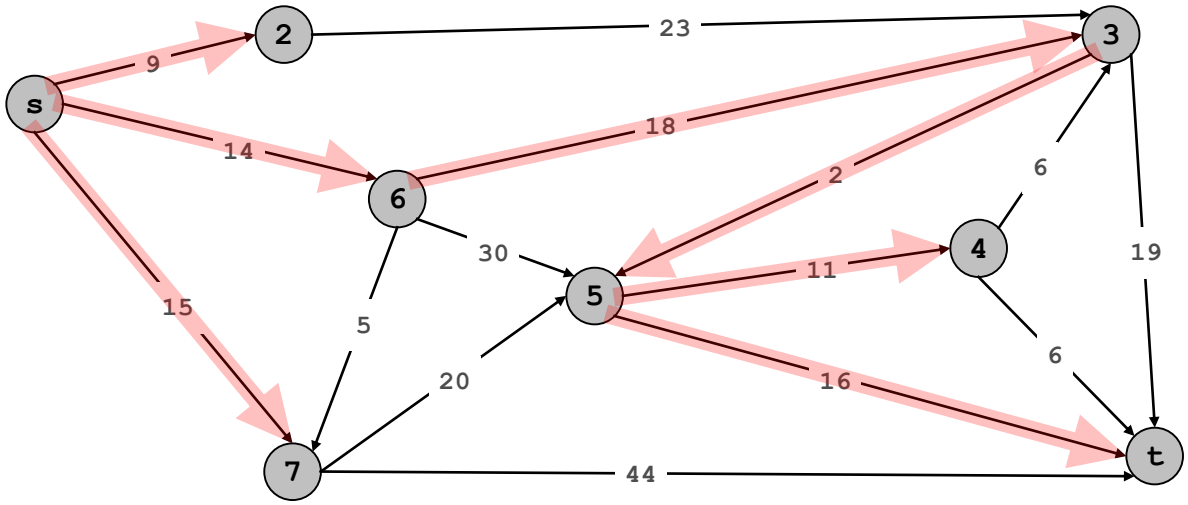
Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Shortest Path Tree

Shortest Path Tree (SPT).

- A rooted tree with root s.
- P(v)=u where s,…, u, v is a shortest path from s to v.
- If there are more than one shortest path from s to v, select one and define the parent of v based on it.

Single Source Shortest Path problem (SSSP): for given s, compute SPT.



Dijkstra's Algorithm

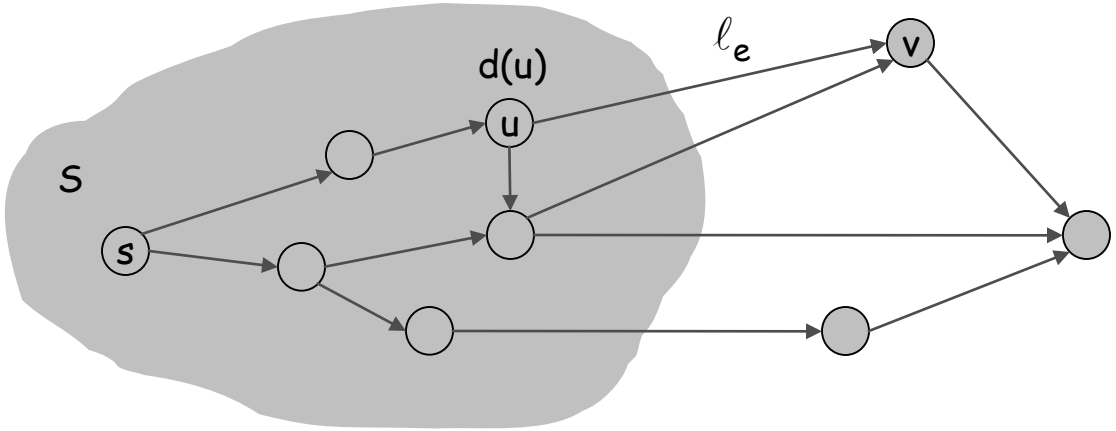
Dijkstra's algorithm. Compute SPT step by step

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm

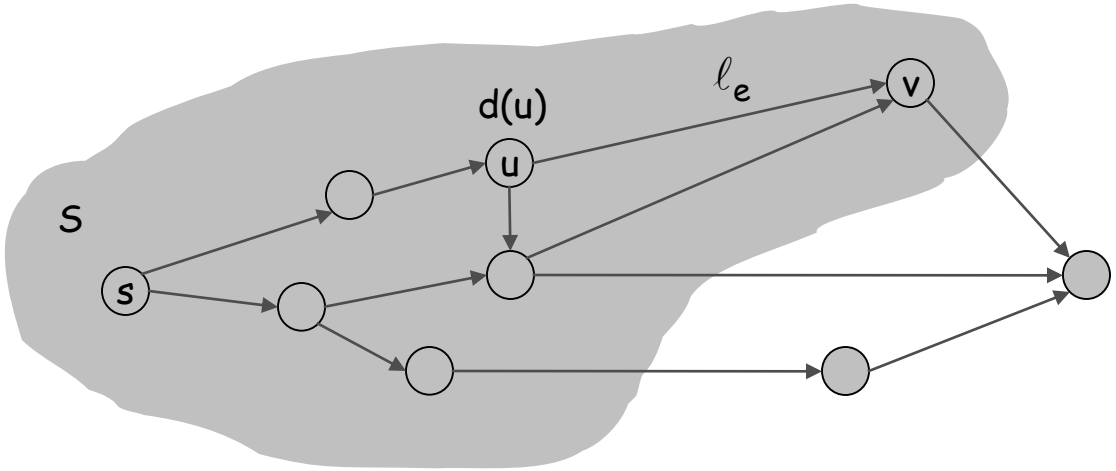
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm: Proof of Correctness

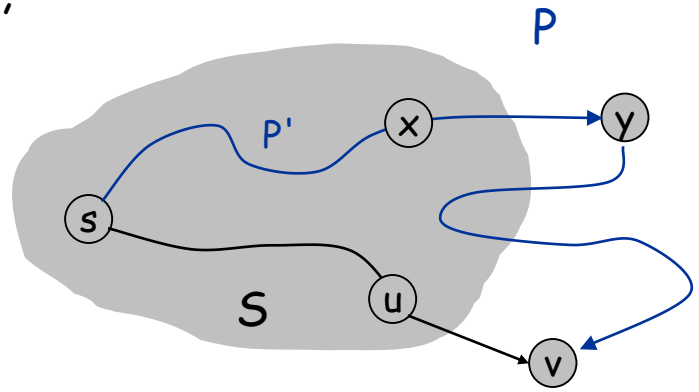
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- $\geq$: nonnegative weights
- $\geq$: inductive hypothesis
- $\geq$: defn of $\pi(y)$
- $\geq$: Dijkstra chose v instead of y

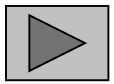
Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{\ \pi(w),\ \pi(v) + \ell_e\ \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

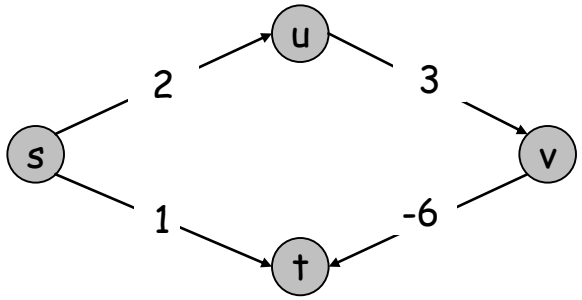


PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

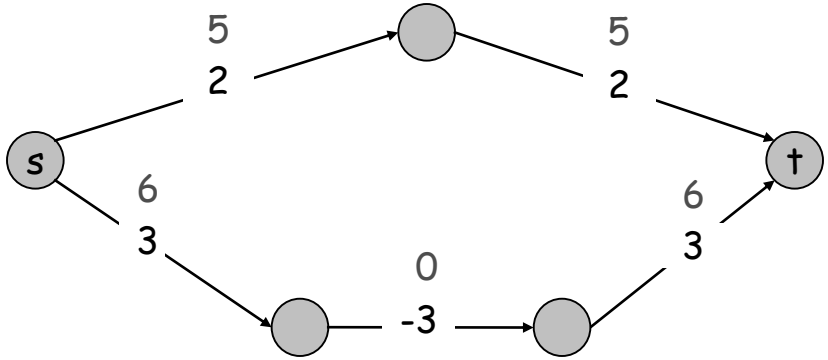
† Individual ops are amortized bounds

Shortest Path: Negative Weights

Dijkstra. Can fail if negative edge costs.

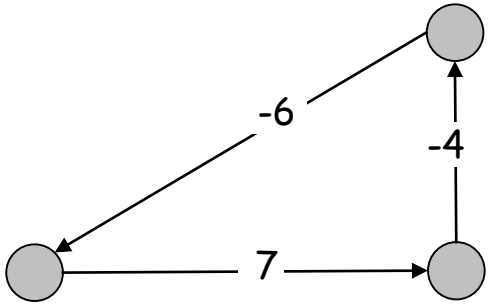


Re-weighting. Adding a constant to every edge weight can fail.

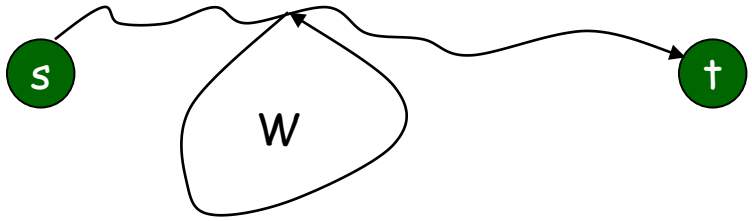


Shortest Paths: Negative Cost Cycles

Negative cost cycle.



Observation. If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple.



$c(W) < 0$

Shortest Paths: Dynamic Programming

Def. OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most i-1 edges.
  - OPT(i, v) = OPT(i-1, v)

- Case 2: P uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min\left\{ OPT(i-1, v),\ \min_{(v, w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

Remark. By previous observation, if no negative cycles, then OPT(n-1, v) = length of shortest v-t path.

Shortest Paths: Implementation

```
Shortest-Path(G, t) {
    foreach node v ∈ V
        M[0, v] ← ∞
    M[0, t] ← 0

    for i = 1 to n-1
        foreach node v ∈ V
            M[i, v] ← M[i-1, v]
        foreach edge (v, w) ∈ E
            M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }
}
```

Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space.

Finding the shortest paths. Maintain a "successor" for each table entry.

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array M[v] = shortest v-t path that we have found so far.
- No need to check edges of the form (v, w) unless M[w] changed in previous iteration.

Theorem. Throughout the algorithm, M[v] is length of some v-t path, and after i rounds of updates, the value M[v] is no larger than the length of shortest v-t path using $\leq$ i edges.

Overall impact.

- Memory: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V {
      M[v] ← ∞
      successor[v] ← ϕ
   }

   M[t] = 0
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               successor[v] ← w
            }
         }
      }
      If no M[w] value changed in iteration i, stop.
   }
}
```

Negative Cycles in a Graph

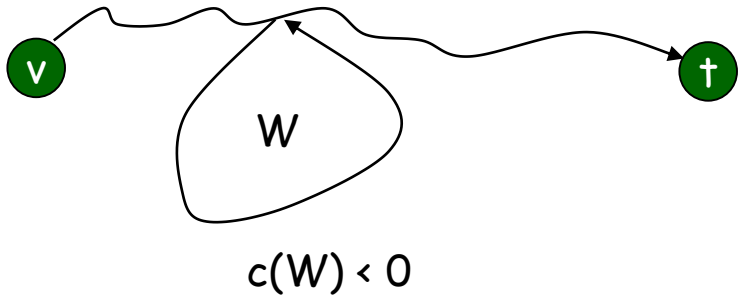
Detecting Negative Cycles

Lemma. If OPT(n,v) = OPT(n-1,v) for all v, then no negative cycles.
Pf. Bellman-Ford algorithm.

Lemma. If OPT(n,v) < OPT(n-1,v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover W has negative cost.

Pf. (by contradiction)

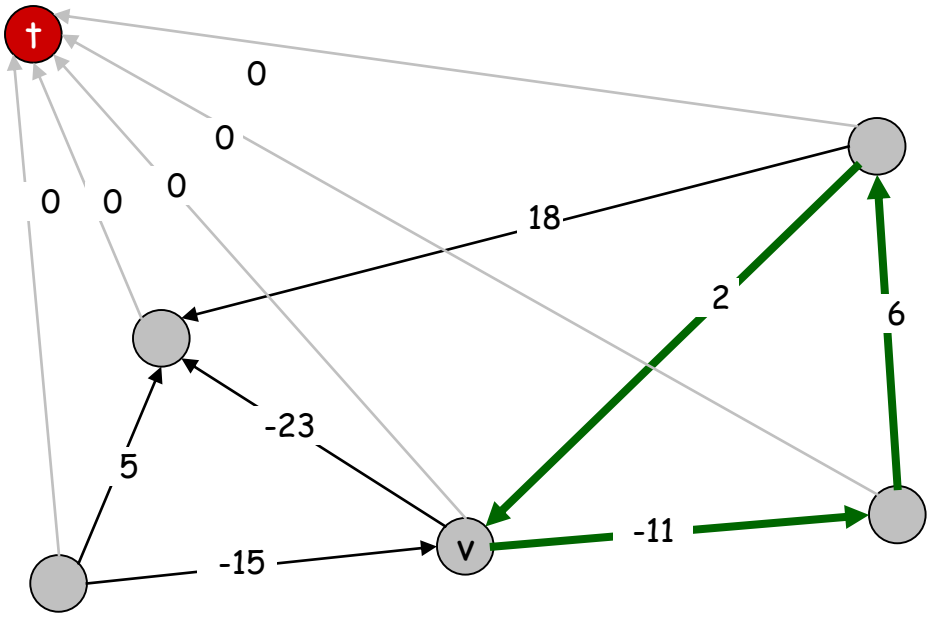
- Since OPT(n,v) < OPT(n-1,v), we know P has exactly n edges.
- By pigeonhole principle, P must contain a directed cycle W.
- Deleting W yields a v-t path with < n edges $\Rightarrow$ W has negative cost.



Detecting Negative Cycles

Theorem. Can detect negative cost cycle in O(mn) time.

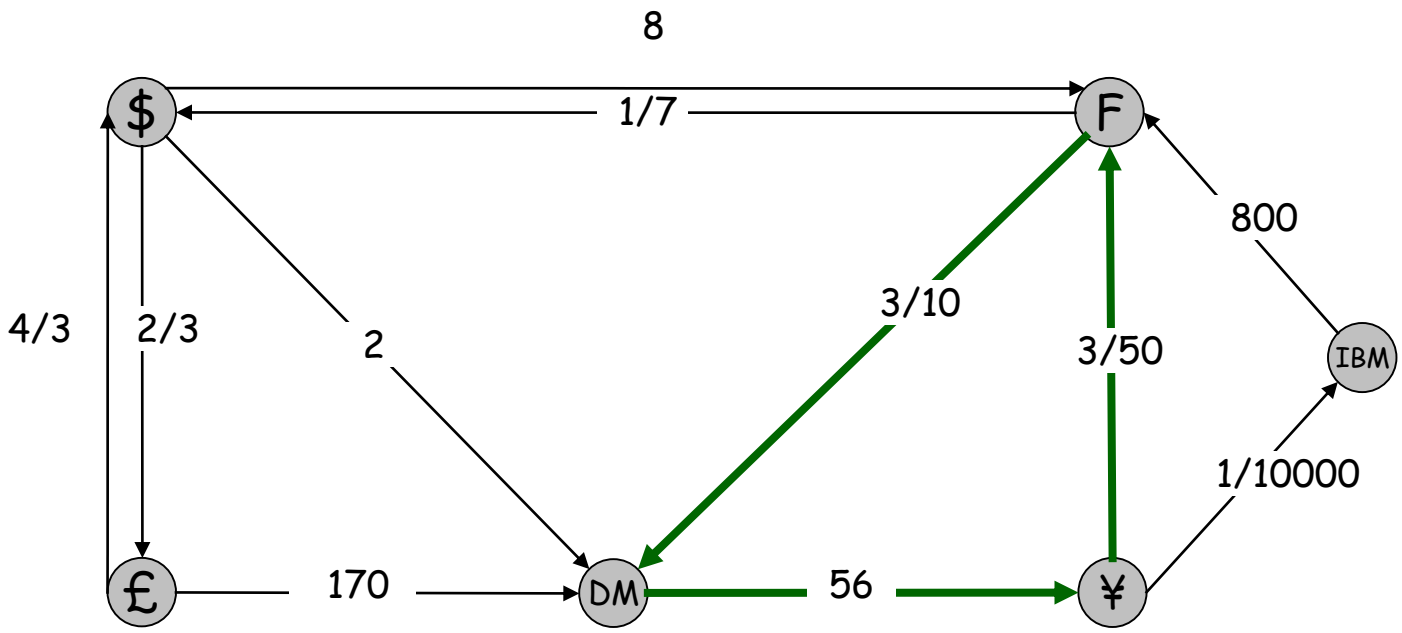
- Add new node t and connect all nodes to t with 0-cost edge.
- Check if OPT(n, v) = OPT(n-1, v) for all nodes v.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from v to t



Detecting Negative Cycles: Application

Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark. Fastest algorithm very valuable!



Detecting Negative Cycles: Summary

Bellman-Ford. $O(mn)$ time, $O(m + n)$ space.

- Run Bellman-Ford for n iterations (instead of n-1).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p. 304 for improved version and early termination rule.

References

References

- Sections 3.1-2, 3.4-6, 4.4, 6.8 and 6.10 of the text book "algorithm design" by Jon Kleinberg and Eva Tardos
- The original slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.