香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Graph Algorithms

Graph Algorithms

- Graph search algorithms
- Connected components in directed/undirected graphs
- Tarjan's Algorithm
- DAGs and Topological orders

Basic Definitions

Definition

- A graph: a group of vertices and edges that are used to connect these vertices

- Definition: A graph G can be defined as an ordered set $G(V, E)$
  - V represents the set of vertices/nodes
  - E represents the set of edges which are used to connect V

Sequential Representation

- Use adjacency matrix to store the mapping represented by vertices and edges

- In adjacency matrix, the rows and columns are represented by the graph vertices

- For a graph having n vertices, the adjacency matrix will have a dimension $n \times n$

Linked Representation

- An adjacency list is used to store the Graph into the computer's memory

- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node

- If all the adjacent nodes are traversed, then store the NULL in the pointer field of last node of the list

Terminology

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Path: a sequence of edges connecting initial node v_0 to terminal node v_n
- Closed Path: A path where the initial node is same as terminal node, i.e., $v_0 = v_n$
- Simple Path: all the nodes of the path are distinct, with the exception $v_0 = v_n$
- Closed Simple Path: a simple path with $v_0 = v_n$
- Cycle: a path which has no repeated edges or vertices except the first and last vertices
- Adjacent Nodes: two nodes u and v are connected via an edge e
  - the nodes u and v are also called as neighbors
- Degree of a Node: the number of edges that are connected with the node
  - A node with degree 0 is called as isolated node

Terminology

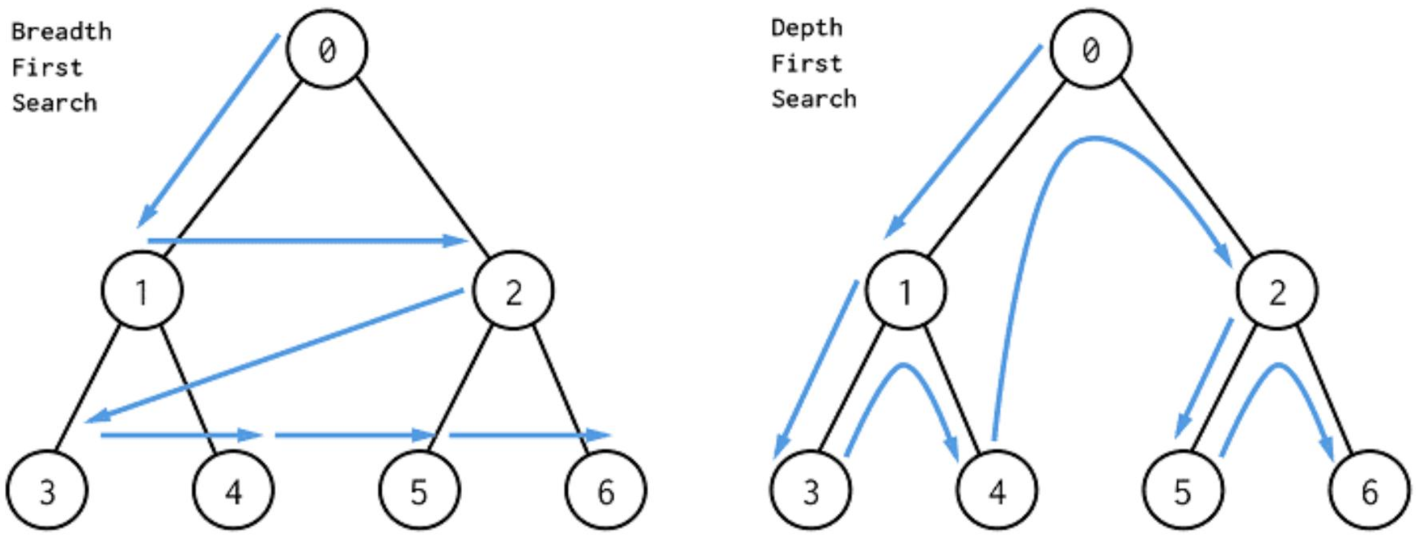
- Connected Graph: a graph in which a path exists between every two vertices u and v in V
  - There are no isolated nodes in connected graph
- Complete Graph: a graph in which there is an edge between each pair of vertices
  - A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph
- Weighted Graph: each edge is assigned with some data such as length or weight
  - The weight of an edge e, $w(e)$, must be positive indicating the cost of traversing the edge
- Digraph: each edge of the graph is associated with some direction
  - The traversing can be done only in the specified direction

Graph Traversal

Connectivity

- s–t connectivity problem: Given two nodes s and t, is there a path between s and t?
- s–t shortest path problem: Given two nodes s and t, what is the length of a shortest path between s and t?
- Applications
  - Friendster
  - Maze traversal
  - Kevin Bacon number
  - Fewest hops in a communication network

Graph Traversal



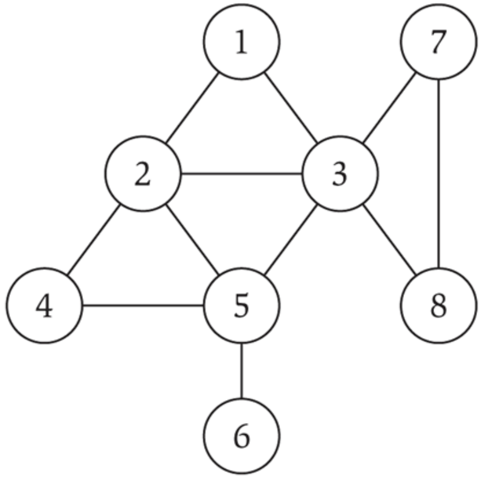
- Traversing the graph means examining all the nodes and vertices of the graph
- Two standard methods to traverse graphs
  - Breadth First Search
  - Depth First Search

Depth First Search (Stack-Based)

- DFS: starts with the initial node, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.
  - Step 1: SET STATUS as UNVISITED (ready state) for each node in G
  - Step 2: Push the starting node A on the stack
  - Step 3: Repeat Steps 4 and 5 until STACK is empty
  - Step 4: Pop the top node N. If node N is VISITED, repeat Step 4; Otherwise, process it and set its STATUS as VISITED (processed state)
  - Step 5: Push on the stack all the neighbours of N with STATUS UNVISITED
  - Step 6: EXIT

Depth First Search (Recursion-Based)


```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```



Assume that we follow neighbours with smaller ID first

DFS-recursive(G, 1) = [1, 2, 4, 5, 6, **3**, 7, 8]

Time complexity $O(n+m)$, when implemented using an adjacency list.

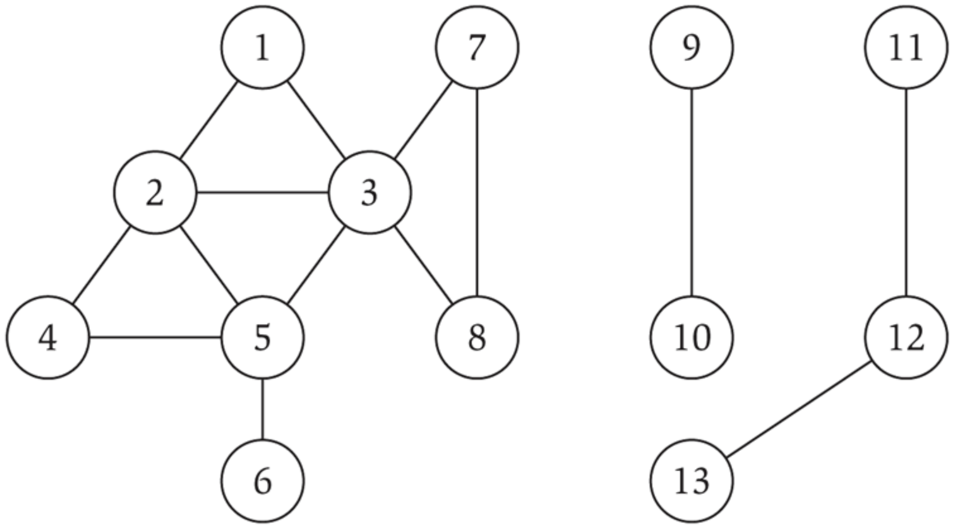
Breadth First Search

- BFS: starts traversing the graph from root node and explores all the neighbours. Then, it selects the nearest node and explore all the unexplored nodes. It follows the same process for each of the nearest node until it finds the goal.
  - Step 1: SET STATUS = 1 (ready state) for each node in G
  - Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
  - Step 3: Repeat Steps 4 and 5 until QUEUE is empty
  - Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
  - Step 5: Enqueue all neighbours of N in the ready state (STATUS = 1) and set their STATUS = 2
  - Step 6: EXIT

Graph Connectivity

Connected Component

- Connected component: find all nodes reachable from s

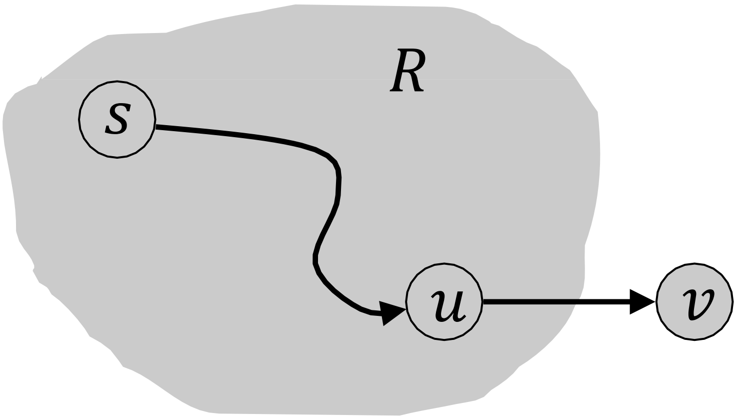


Connected component containing node 1 is [1, 2, 3, 4, 5, 6, 7, 8]

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 ∈ R and v ∉ R
  Add v to R
Endwhile
```



it's safe to add v

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

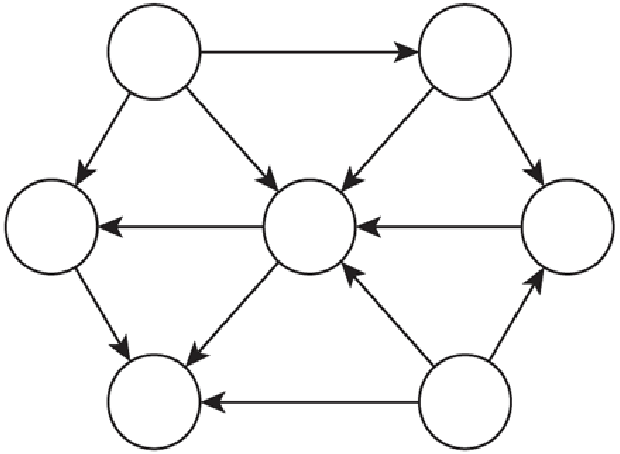
- BFS = explore in order of distance from s
- DFS = explore in a different way

Connectivity in Directed Graphs

Directed Graph

Notation: $G = (V, E)$

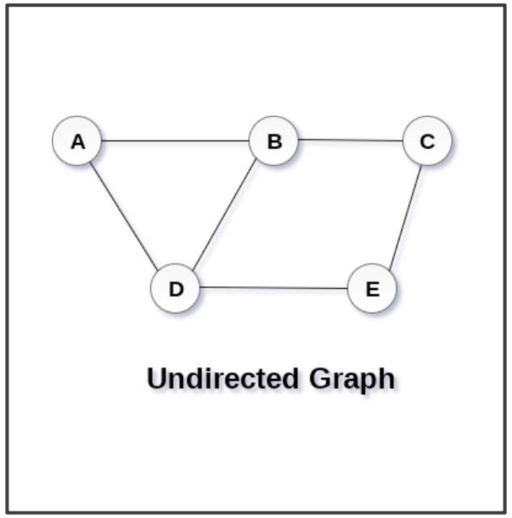
- Edge (u, v) leaves node u and enters node v

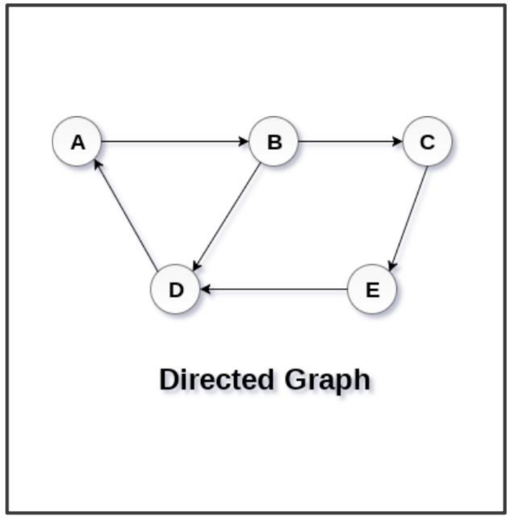


Ex. Web graph: hyperlink points from one web page to another

- Orientation of edges is crucial
- Modern web search engines exploit hyperlink structure to rank web pages by importance

Undirected V.S. Directed





- In an undirected graph, edges are not associated with the directions with them
  - If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B
- In a directed graph, edges form an ordered pair
  - Edges represent a specific path from some vertex A to another vertex B
  - Node A is called initial node while node B is called terminal node

Strong Connectivity

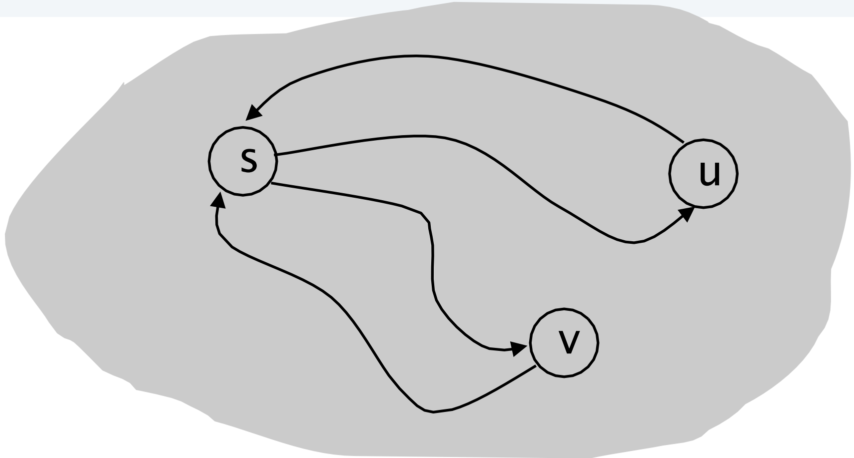
- Def. Nodes u and v are mutually reachable if there is both a path from u to v and also a path from v to u
- Def. A graph is strongly connected if every pair of nodes is mutually reachable
- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node

Pf. $\Rightarrow$ Follows from definition

Pf. $\Leftarrow$ Path from u to v: concatenate $u \rightsquigarrow s$ path with $s \rightsquigarrow v$ path

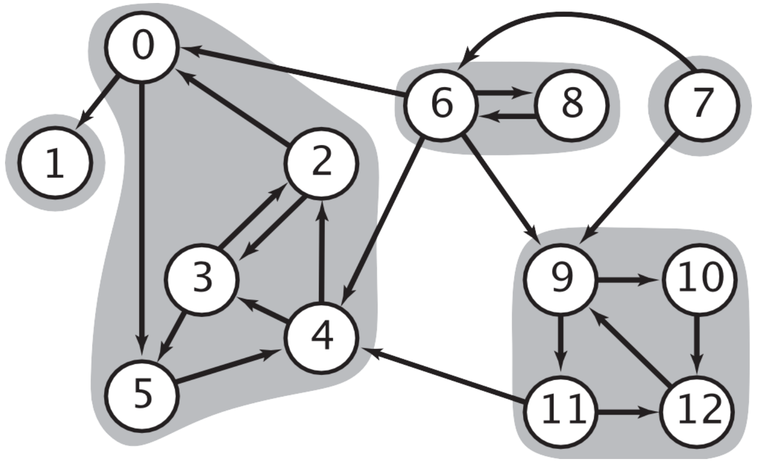
Path from v to u: concatenate $v \rightsquigarrow s$ path with $s \rightsquigarrow u$ path ▪

ok if paths overlap

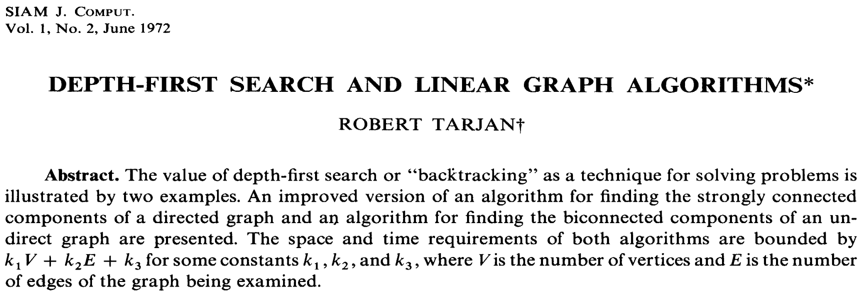


Strong Components

- Def. A strong component is a maximal subset of mutually reachable nodes



Theorem. [Tarjan 1972] Can find all strong components in $O(m+n)$ time


SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Tarjan's Algorithm: Overview

1. **Initialization**:
   1. Assign a unique index to each node, initialize as undefined
   2. Assign a lowlink value to each node, initialize as undefined
   3. Create an empty stack to keep track of nodes in the current search path
   4. Create an empty list to store the strongly connected components (SCCs)
2. **Depth-First Search (DFS) Loop**:
   1. For each node v in the graph:
      1. If v has not been visited:
         1. Call the strongConnect function on v
3. **strongConnect Function**:
   1. Set the index of v to the current global index
   2. Set the lowlink of v to the current global index
   3. Push v onto the stack
   4. Mark v as being on the stack
   5. Increment the global index
4. **Explore Adjacent Nodes**:
   1. For each adjacent node u of v:
      1. If u has not been visited:
         1. Recursively call strongConnect(u)
         2. Update the lowlink of v to the minimum of v.lowlink and u.lowlink
      2. If u is on the stack:
         1. Update the lowlink of v to the minimum of v.lowlink and u.index
5. **Identify SCC**:
   1. If the lowlink of v is equal to its index:
      1. Pop nodes from the stack until v is popped
      2. Each popped node is part of a new SCC
      3. Add the popped nodes to the list of SCCs
6. **Output**:
   1. After all nodes have been processed, the list of SCCs contains all the strongly connected components of the graph

Tarjan's Algorithm: Pseudocode

```
// GLOBAL VARIABLES
//     num <- global array of size V initialized to -1
//     lowest <- global array of size V initialized to -1
//     visited <- global array of size V initialized to false
//     processed <- global array of size V initialized to false
//     s <- global empty stack
//     i <- 0
```

```
algorithm TarjanAlgorithm(G):
    // INPUT
    //     G = the graph
    // OUTPUT
    //     SCCs of G are found

    visted <- an empty global visited map
    for v in G.V:
        if visited[v] = false:
            // global variables are accessible from within DFS
            DFS(G, v)
```

```
algorithm DFS(G, v):
    // INPUT
    //     G = the graph
    //     v = the current vertex
    // OUTPUT
    //     Vertices reachable from v are processed, their SCCs are reported

    num[v] <- i
    lowest[v] <- num[v]
    i <- i + 1
    visited[v] <- true
    s.push(v)

    for u in G.neighbours[v]:
        if visited[u] = false:
            DFS(G, u)
            lowest[v] <- min(lowest[v], lowest[u])
        else if processed[u] = false:
            lowest[v] <- min(lowest[v], num[u])

    processed[v] <- true

    if lowest[v] = num[v]:
        scc <- an empty set
        sccVertex <- s.pop()

        while sccVertex != v:
            scc.add(sccVertex)
            sccVertex <- s.pop()

        scc.add(sccVertex)

        Process the found scc in the desired way

    return
```

Tarjan's Algorithm

- Tarjan's algorithm is a modification of the DFS traversal. Hence, the complexity of the algorithm is linear: $O(n + m)$
  - To achieve the mentioned complexity, we must use the adjacency list representation of the graph

- Tarjan's algorithm for finding strongly connected components in directed graphs. It's an optimal linear time algorithm

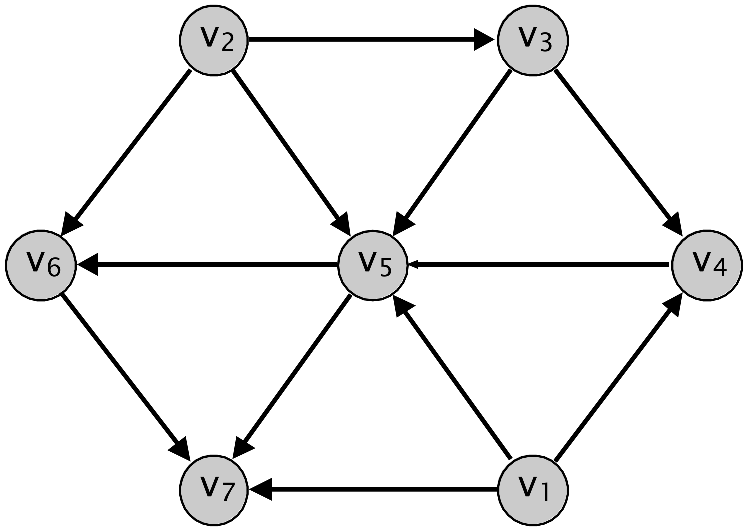
- More Tarjan's algorithms, have a try if you are interested!

DAG & Topological Ordering

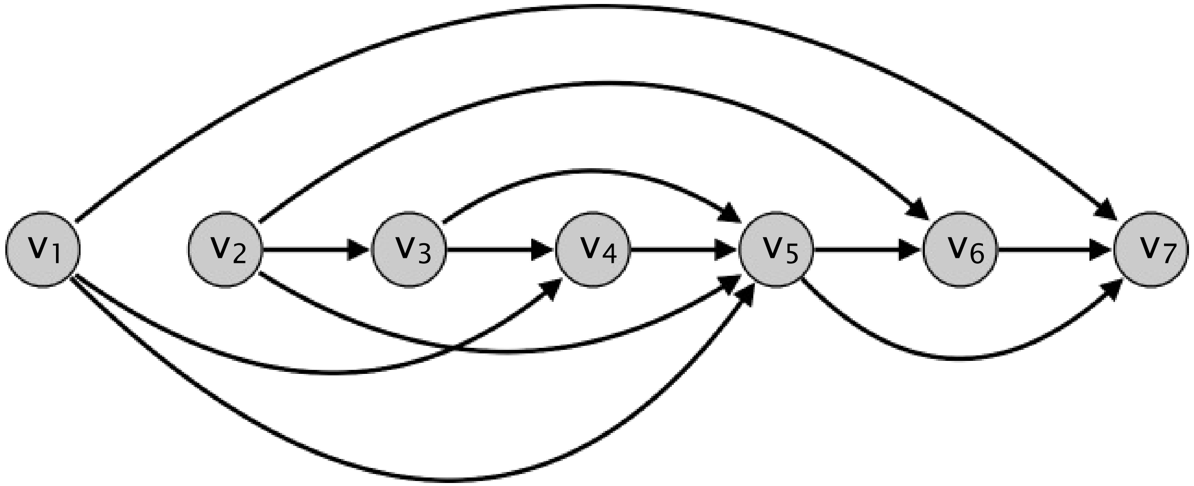
Directed Acyclic Graphs

- Def. A DAG is a directed graph that contains no directed cycles

- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \dots, v_n$ so that for every edge (v_i, v_j) we have $i < j$

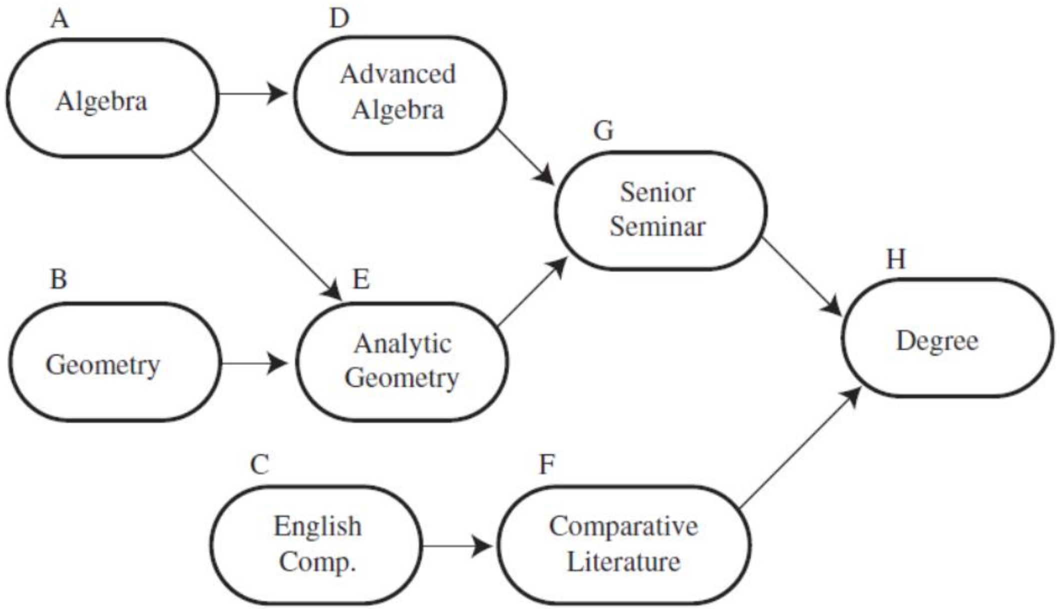


a DAG



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

Topological Sorting Algorithm

- 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
    - *decrease count(*w*) for all edges from* v *to* w*; and add* w *to* S *if count(*w*) hits* 0
    - *this is O(1) per edge* ▪
- Topological-sort cannot handle graphs with cycles!

Network Flow

Network Flow

- Ford-Fulkerson algorithm.
- Edmonds-Karp Algorithm
- Max-Flow & Min-Cut.

Maximum Flow Problem

- Definition (Value)

The value v(f) of a flow f is $f^{\text{out}}(s)$.

  – That is: it is the amount of material that leaves s.

- **Maximum Flow Problem**

  Given a flow network G, find a flow f of maximum possible value.

Ford-Fulkerson Algorithm

Residual Graph

- We define a residual graph G_f. G_f depends on some flow f:
  - G_f contains the same nodes as G.
  - Forward edges: For each edge e = (u,v) of G for which $f(e)<c_e$, include an e' = (u,v) in G_f with capacity $c_e - f(e)$.
  - Backward edges: For each edge e = (u,v) in G with $f(e)>0$, we include an e' = (v,u) in G_f with capacity f(e).

Augmenting Paths

- Let P be an s–t path in the residual graph G_f.
- Let bottleneck(P, f) be the smallest capacity in G_f on any edge of P.
- If bottleneck(P, f) > 0 then we can increase the flow by sending bottleneck(P, f) along the path P.

Augmenting Paths

```
augment(f, P):
    b = bottleneck(P,f)
    For each edge (u,v) ∈ P:
        If e = (u,v) is a forward edge:
            Increase f(e) in G by b    //add some flow
        Else:
            e' = (v,u)
            Decrease f(e') in G by b  //erase some flow
        EndIf
    EndFor
    Return f
```

Ford-Fulkerson Algorithm

```
MaxFlow(G):
  // initialize:
  Set f[e] = 0 for all e in G

  // while there is an s-t path in Gf:
  While P = FindPath(s,t, Residual(G,f)) != None:
     f = augment(f, P)
     UpdateResidual(G, f)
  EndWhile
  Return f
```

Running Time

- At every step, the flow values f(e) are integers. Start with integers and always add or subtract integers
- At every step we increase the amount of flow v(f) sent by at least 1 unit.
- We can never send more than C := $\sum_{e \text{ leaving } s} c_e$.
- Theorem: The Ford-Fulkerson algorithm terminates in C iterations of the *While* loop.

Time in the While loop

- If G has m edges, G_f has ≤ 2m edges.
- Can find an s–t path in G_f in time O(m+n) time with DFS or BFS.
- Since $m \geq n/2$ (every node is adjacent to some edge), $O(m + n) = O(m)$.

Theorem: The Ford-Fulkerson algorithm runs in O(mC) time.

Caveats, etc.

Note this is pseudo-polynomial because it depends on the size of the integers in the input.

You can remove this with slightly different algorithms. E.g.:

– $O(nm^2)$: Edmonds-Karp algorithm (use BFS to find the augmenting path)

– $O(m^2 \log C)$ or

– $O(n^2 m)$ or $O(n^3)$

Edmonds-Karp Algorithm

Edmonds-Karp Algorithm

- Follow the same procedure as Ford-Fulkerson, except that we find the shortest augmenting path each time (on the residual graph).

- When finding paths, regard the residual graph as unweighted.

- Time complexity: $O(nm^2)$. (m is #edges; n is #vertices.)

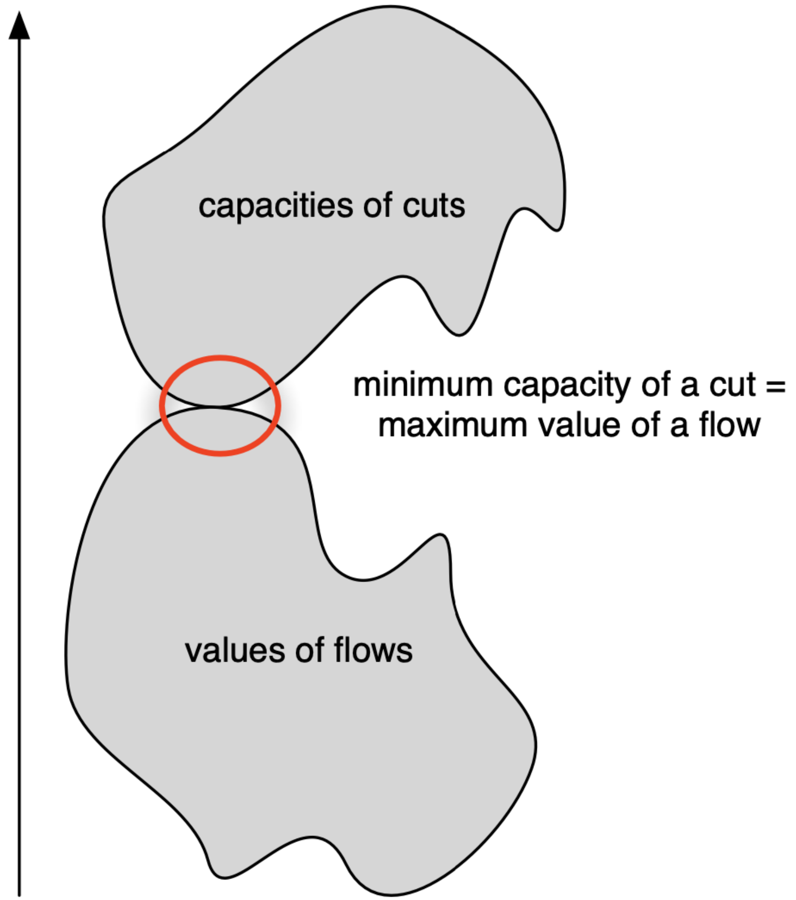
Time Complexity Analysis

- m: number of edges.
- n: number of vertices.
- Each iteration has $O(m)$ time complexity.
- The number of iterations is at most $m \cdot n$.
- The worst-case time complexity is $O(nm^2)$.

Prove by yourself with ref2 if you are interested

Max-Flow & Min-Cut

Max-Flow = Min-Cut



Max-Flow = Min-Cut

Therefore,

- v(f*)=capacity(A*,B*).
- No flow can have value bigger than capacity(A*,B*).
- So, f* must be an maximum flow.
- And (A*,B*) has to be a minimum-capacity cut.

Theorem (Max-flow = Min-cut): The value of the maximum flow in any flow graph is equal to the capacity of the minimum cut.

Reference

- L. R. Ford and D. R. Fulkerson. Flows in Networks. Princeton University Press, 1962.

Finding the Min-Capacity Cut

Our proof that maximum flow = minimum cut can be used to actually find the minimum capacity cut:

- Find the maximum flow f*.
- Construct the residual graph G_{f*} for f*. With backward edges
- Do a BFS to find the nodes reachable from s in G_{f*}. Let the set of these nodes be called A*.
- Let B* be all other nodes.
- Return (A*, B*) as the minimum capacity cut.