CSE 548: Algorithms

Basic Graph Algorithms

R. Sekar

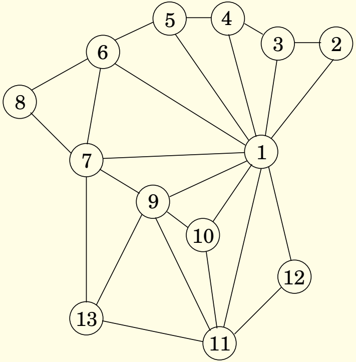
Overview

- Graphs provide a concise representation of a range problems

  Map coloring – more generally, resource contention problems

  Networks — communication, traffic, social, biological, ...

(b)



Definition and Representations

Definition

- *A* graph $G = (V, E)$*, where* V *is a set of vertices, and* E *a set of edges.*
- *An edge* e *of the form* (v_1, v_2) *is said to span vertices* v_1 *and* v_2*.*
- *The edges in a* directed graph *are directed.*

Definition and Representations

Definition

- *A graph* $G = (V, E)$*, where* V *is a set of vertices, and* E *a set of edges.*
- *An edge* e *of the form* (v_1, v_2) *is said to span vertices* v_1 *and* v_2*.*
- *The edges in a* directed graph *are directed.*
- *A* $G' = (V', E')$ *is called a subgraph of* G *if* $V' \subseteq V$ *and* $E' = \{(u, v) \in E | u, v \in V'\}$.

Definition and Representations

Definition

- *A graph* $G = (V, E)$*, where* V *is a set of vertices, and* E *a set of edges.*
- *An edge* e *of the form* (v_1, v_2) *is said to span vertices* v_1 *and* v_2*.*
- *The edges in a* directed graph *are directed.*
- *A* $G' = (V', E')$ *is called a subgraph of* G *if* $V' \subseteq V$ *and* $E' = \{(u, v) \in E | u, v \in V'\}$*.*

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix a, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$

Definition and Representations

Definition

- *A graph* $G = (V, E)$*, where* V *is a set of vertices, and* E *a set of edges.*
- *An edge* e *of the form* (v_1, v_2) *is said to span vertices* v_1 *and* v_2*.*
- *The edges in a* directed graph *are directed.*
- *A* $G' = (V', E')$ *is called a subgraph of* G *if* $V' \subseteq V$ *and* $E' = \{(u, v) \in E | u, v \in V'\}$*.*

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix a, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$

Adjacency list

Each vertex v is associated with a linked list consisting of all vertices u such that $(v, u) \in E$.

Definition and Representations

Definition

- *A graph $G = (V, E)$, where V is a set of vertices, and E a set of edges.*
- *An edge e of the form (v_1, v_2) is said to span vertices v_1 and v_2.*
- *The edges in a* directed graph *are directed.*
- *A $G' = (V', E')$ is called a subgraph of G if $V' \subseteq V$ and $E' = \{(u, v) \in E | u, v \in V'\}$.*

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix a, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$

Adjacency list

Each vertex v is associated with a linked list consisting of all vertices u such that $(v, u) \in E$.

Adjacency matrix uses $O(n^2)$ storage; adjacency list uses $O(|V| + |E|)$ storage.

Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.

Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.

dfs(V, E)

foreach $v \in V$ **do** $visited[v] = false$
foreach $v \in V$ **do**
  if not $visited[v]$ **then** $explore(V, E, v)$

Depth-First Search (DFS)

- A technique for traversing all vertices in the graph.
- Very versatile, forms the linchpin of many graph algorithms.

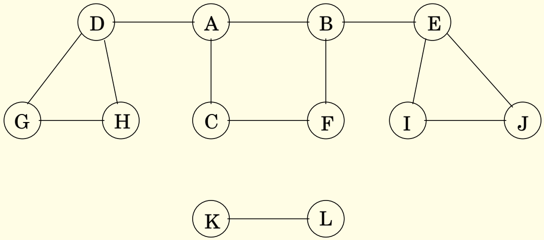
dfs(V, E)

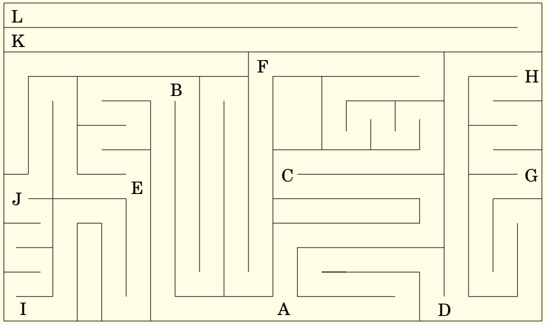
foreach $v \in V$ **do** $visited[v] = false$
foreach $v \in V$ **do**
  if not $visited[v]$ **then** $explore(V, E, v)$

explore(V, E, v)

$visited[v] = true$
$previsit(v)$ /*A placeholder for now*/
foreach $(v, u) \in E$ **do**
  if not $visited[u]$ **then** $explore(V, E, u)$
$postvisit(v)$ /*Another placeholder*/

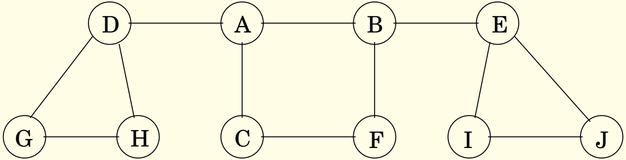
Graphs, Mazes and DFS

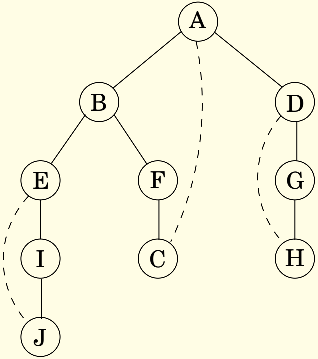




If a maze is represented as a graph, then DFS of the graph amounts to an exploration and mapping of the maze.

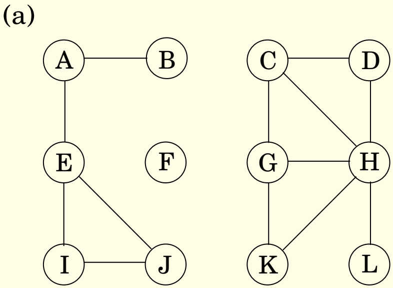
A graph and its DFS tree

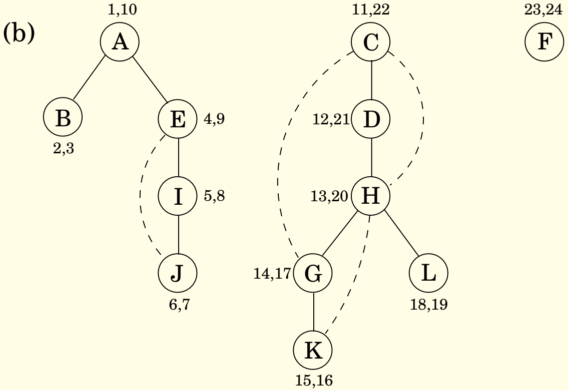




DFS uses $O(|V|)$ space and $O(|E| + |V|)$ time.

DFS and Connected Components





A *connected component* of a graph is a maximal subgraph where there is path between any two vertices in the subgraph, i.e., it is a maximal *connected subgraph*.

DFS Numbering

Associate post and pre numbers with each visited node by defining *previsit* and *postvisit*

$previsit(v)$

$pre[v] = clock$

$clock{+}{+}$

DFS Numbering

Associate post and pre numbers with each visited node by defining *previsit* and *postvisit*

$previsit(v)$

$pre[v] = clock$

$clock{+}{+}$

$postvisit(v)$

$post[v] = clock$

$clock{+}{+}$

DFS Numbering

Associate post and pre numbers with each visited node by defining *previsit* and *postvisit*

$previsit(v)$

$pre[v] = clock$

$clock{+}{+}$

$postvisit(v)$

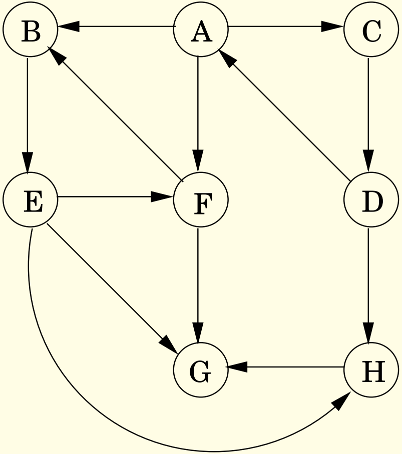
$post[v] = clock$

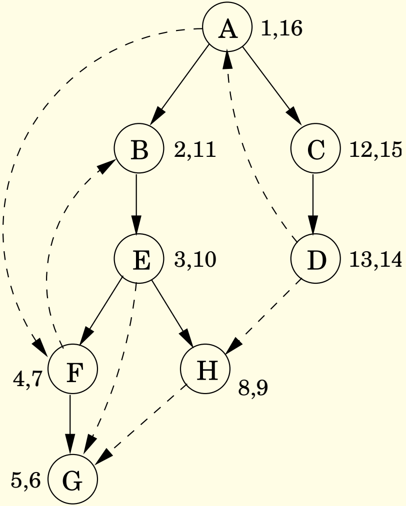
$clock{+}{+}$

Property

For any two vertices u and v, the intervals $[pre[u], post[u]]$ and $[pre[v], post[v]]$ are either disjoint, or one is contained entirely within another.

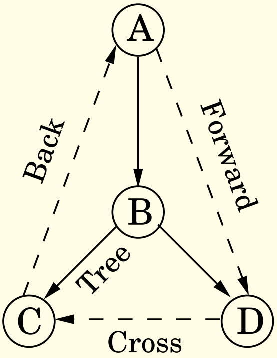
DFS of Directed Graph





DFS and Edge Types

DFS tree



`pre/post` *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

No cross edges in undirected graphs!

Back and forward edges merge

Directed Acyclic Graphs (DAGs)

A directed graph that contains no cycles.

Often used to represent (acyclic) dependencies, partial orders,...

Property (DAGs and DFS)

- *A directed graph has a cycle iff its DFS reveals a back edge.*
- *In a dag, every edge leads to a vertex with lower post number.*
- *Every dag has at least one source and one sink.*

Topological Sort

A way to linearize DAGs while ensuring that for every vertex, all its ancestors appear before itself.

Applications: spreadsheet recomputation of formulas, Make (and other compile/build systems) and Task scheduling/project management.

Topological Sort

topoSort(V, E)

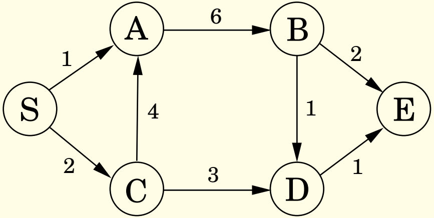
while $|V| \neq 0$

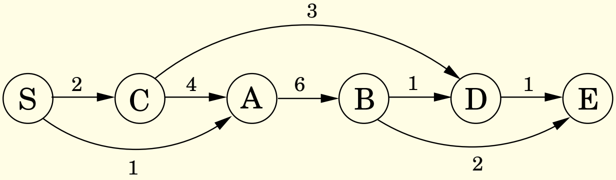
if there is a vertex v in V with in-degree of 0

output v

$V = V - \{v\}; E = E - \{e \in E | e \text{ is incident on } v\})$

else output "graph is cyclic"; **break**





Topological Sort

topoSort(V, E)

while $|V| \neq 0$

if there is a vertex v in V with in-degree of 0

output v

$V = V - \{v\}; E = E - \{e \in E | e \text{ is incident on } v\})$

else output "graph is cyclic"; **break**

Correctness:

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs v, it has already output v's ancestors

Topological Sort

topoSort(V, E)

while $|V| \neq 0$
  if there is a vertex v in V with in-degree of 0
    output v
    $V = V - \{v\}; E = E - \{e \in E | e \text{ is incident on } v\})$
  else output "graph is cyclic"; **break**

Correctness:

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs v, it has already output v's ancestors

Performance: What is the runtime? Can it be improved using DFS properties of DAGs?

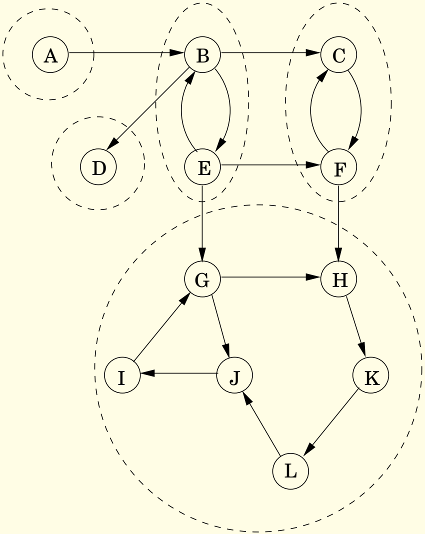
Strongly Connected Components (SCC)

For directed graphs, SCCs are the equivalent of connected components in undirected graphs.

Definition (SCC)

- *Two vertices u and v in a directed graph are connected if there is a path from u to v and vice-versa.*
- *A directed graph is strongly connected if any pair of vertices in the graph are connected.*
- *A subgraph of a directed graph is said to be an SCC if it is a maximal subgraph that is strongly connected.*

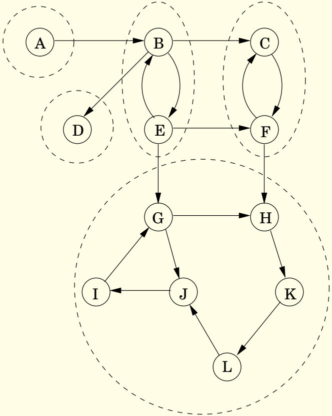
SCC Example



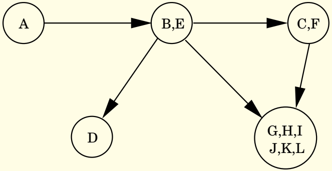
DAG of SCCs

Property

Every directed graph is a dag of its strongly connected components.



(b)



Towards an Algorithm for Computing SCCs

- Pick a sink SCC.
- Output all nodes in this SCC.
  - We can just do a DFS starting from any node v in this SCC!
    - Because this is a sink SCC, this DFS cannot reach any other SCC, so will only output this SCC.
- Delete these nodes from the graph and repeat the whole process.

Towards an Algorithm for Computing SCCs

- Pick a sink SCC.
- Output all nodes in this SCC.
  - We can just do a DFS starting from any node v in this SCC!
    - Because this is a sink SCC, this DFS cannot reach any other SCC, so will only output this SCC.
- Delete these nodes from the graph and repeat the whole process.

But how do we find a node in the sink SCC?

Towards an Algorithm for Computing SCCs

Property

- *When explore(u) returns, it has visited all (and only) the nodes reachable from u.*
- *If C and C' are SCCs and there is an edge from C to C' then:*
  the highest post number in C will be larger than the highest post number in C'.

Towards an Algorithm for Computing SCCs

Property

- *When explore(u) returns, it has visited all (and only) the nodes reachable from u.*
- *If C and C' are SCCs and there is an edge from C to C' then:*
  the highest post number in C will be larger than the highest post number in C'.

Corollary

The node that receives the highest post number after DFS must be in a source SCC.

Towards an Algorithm for Computing SCCs

Property

- *When explore(u) returns, it has visited all (and only) the nodes reachable from u.*
- *If C and C' are SCCs and there is an edge from C to C' then: the highest post number in C will be larger than the highest post number in C'.*

Corollary

The node that receives the highest post number after DFS must be in a source SCC.

Property

For a graph G, let G_R denote the graph formed by reversing every edge in G. Then

- *The SCCs of G and G_R are identical.*
- *A source SCC of G_R is a sink SCC of G.*

An Algorithm for Computing SCCs

1. Construct G_R from G by reversing every edge in the given graph G.
2. The node v with the highest post number is in a source SCC of G_R.
   - So, v must be in a sink SCC of G.
3. Invoke $explore(v)$ in G to output this sink SCC.
4. Delete these nodes from G and G_R, and repeat from Step 2.

An Algorithm for Computing SCCs

1. Construct G_R from G by reversing every edge in the given graph G.
2. The node v with the highest post number is in a source SCC of G_R.
   - So, v must be in a sink SCC of G.
3. Invoke $explore(v)$ in G to output this sink SCC.
4. Delete these nodes from G and G_R, and repeat from Step 2.

Can we do all this in linear time?

Breadth-first Search (BFS)

- Traverse the graph by "levels"
  - $BFS(v)$ visits v first
  - Then it visits all immediate children of v
  - then it visits children of children of v, and so on.

Breadth-first Search (BFS)

- Traverse the graph by “levels”
  - $BFS(v)$ visits v first
  - Then it visits all immediate children of v
  - then it visits children of children of v, and so on.
- As compared to DFS, BFS uses a queue (rather than a stack) to remember vertices that still need to be explored

BFS Algorithm

bfs(V, E, s)

foreach $u \in V$ **do** $visited[u] = false$

$q = \{s\}; visited[s] = true$

while q is nonempty **do**

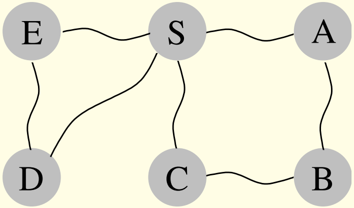
  $u = deque(q)$

  foreach edge $(u, v) \in E$ **do**

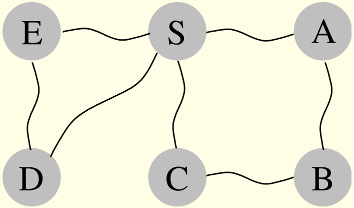
    if not $visited[v]$ **then**

      $queue(q, v); visited[v] = true$

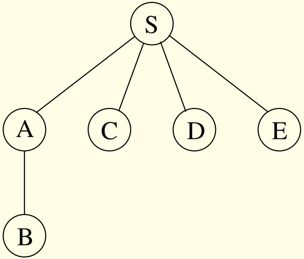
BFS Algorithm Illustration



BFS Algorithm Illustration



Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



Shortest Paths and BFS

BFS automatically computes shortest paths!

bfs(V, E, s)

foreach $u \in V$ **do** $dist[u] = \infty$

$q = \{s\}; dist[s] = 0$

while q is nonempty **do**

  $u = deque(q)$

  foreach edge $(u, v) \in E$ **do**

    if $dist[v] = \infty$ **then**

      $queue(q, v); dist[v] = dist[u] + 1$

Shortest Paths and BFS

BFS automatically computes shortest paths!

$bfs(V, E, s)$

foreach $u \in V$ **do** $dist[u] = \infty$

$q = \{s\}; dist[s] = 0$

while q is nonempty **do**

  $u = deque(q)$

  foreach edge $(u, v) \in E$ **do**

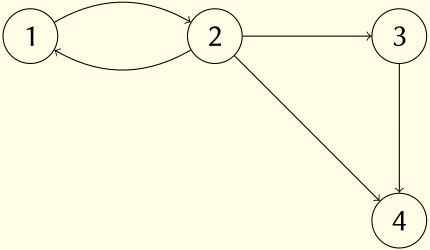
    if $dist[v] = \infty$ **then**

      $queue(q, v); dist[v] = dist[u] + 1$

But not all paths are created equal! We would like to compute shortest weighted path — a topic of future lecture.

Graph paths and Boolean Matrices

A graph and its boolean matrix representation



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Graph paths and Boolean Matrices

- Let A be the adjacency matrix for a graph G, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from v_i to v_j

Graph paths and Boolean Matrices

- Let A be the adjacency matrix for a graph G, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from v_i to v_j
- Let $C = A + B$. Then $C_{ij} = 1$ iff there is path of length ≤ 2 between v_i and v_j

Graph paths and Boolean Matrices

- Let A be the adjacency matrix for a graph G, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from v_i to v_j
- Let $C = A + B$. Then $C_{ij} = 1$ iff there is path of length ≤ 2 between v_i and v_j
- Define $A^* = A^0 + A^1 + A^2 + \cdots$. If $D = A^*$ then $D_{ij} = 1$ iff v_j is reachable from v_i.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Shortest paths and Matrix Operations

- Redefine operations on matrix elements so that $+$ becomes *min*, and $*$ becomes integer addition.

Shortest paths and Matrix Operations

- Redefine operations on matrix elements so that $+$ becomes *min*, and $*$ becomes integer addition.
- $D = A^*$ then $D_{ij} = k$ iff the shortest path from v_j to v_i is of length k