Graphs and graph search (BFS/DFS/SCC)

Lecture 16

Tuesday, October 22, 2024

LaTeXed: August 25, 2024 14:23

16.1
Graph Basics

Why Graphs?

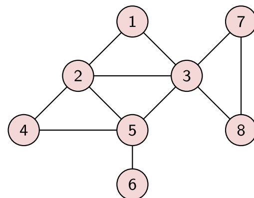
1. Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don't even look like graph problems.
2. Fundamental objects in Computer Science, Optimization, Combinatorics
3. Many important and useful optimization problems are graph problems
4. Graph theory: elegant, fun and deep mathematics

Graph

Definition 16.1.

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$. Use shorthand $uv = \{u, v\}$.



Example 16.2.

In figure, $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

Example: Modeling Problems as Search

State Space Search

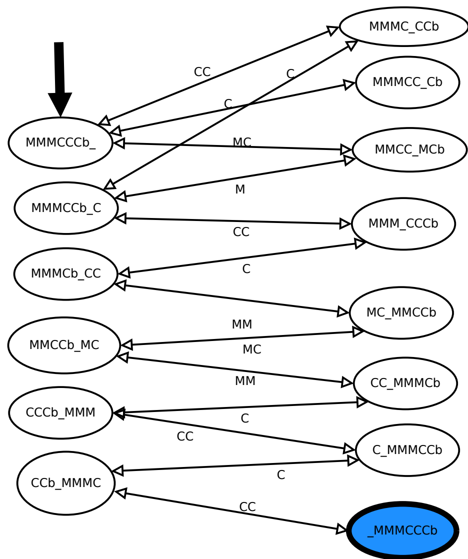
Many search problems can be modeled as search on a graph.
The trick is figuring out what the vertices and edges are.

Missionaries and Cannibals

- Three missionaries, three cannibals, one boat, one river
- Boat carries two people, must have at least one person
- Must all get across
- At no time can cannibals outnumber missionaries

How is this a graph search problem?
What are the vertices?
What are the edges?

Cannibals and Missionaries: Is the language empty?



Problems goes back to 800 CE
Versions with brothers and sisters.
Jealous Husbands.
All bad names to a simple problem...

Problems on DFAs and NFAs sometimes are just problems on graphs

1. M: DFA/NFA is $L(M)$ empty?
2. M: DFA is $L(M) = \Sigma^*$?
3. M: DFA, and a string w. Does M accepts w?
4. N: NFA, and a string w. Does N accepts w?

Exercise

State the following problems as graph problems, and describe an algorithm that solves them (we will solve them later on in the course):

1. M: DFA, is $L(M)$ infinite?
2. N: NFA, is $L(M)$ finite?
3. M: DFA/NFA, compute the shortest word in $L(M)$?
4. M: DFA, if $L(M)$ is finite, compute the longest word $w \in L(M)$?

[Solutions would probably not be recorded for these questions (lack of time).]

16.1.1
Graph notation and representation

Notation and Convention

Notation

An edge in an undirected graphs is an unordered pair of nodes and hence it is a set. Conventionally we use uv for $\{u, v\}$ when it is clear from the context that the graph is undirected.

1. u and v are the end points of an edge $\{u, v\}$
2. Multi-graphs allow
   2.1 loops which are edges with the same node appearing as both end points
   2.2 multi-edges: different edges between same pairs of nodes
3. In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

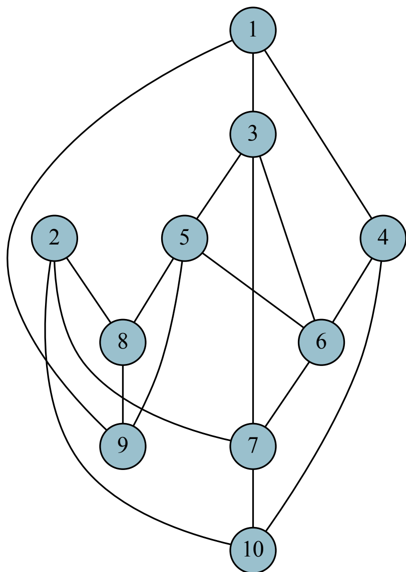
Graph Representation I

Adjacency Matrix

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ adjacency matrix A where

1. $A[i, j] = A[j, i] = 1$ if $\{i, j\} \in E$ and $A[i, j] = A[j, i] = 0$ if $\{i, j\} \notin E$.
2. Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
3. Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$

Graph adjacency matrix example [10 vertices]



	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	0	0	1	0
2	0	0	0	0	0	0	1	1	0	1
3	1	0	0	0	1	1	1	0	0	0
4	1	0	0	0	0	1	0	0	0	1
5	0	0	1	0	0	1	0	1	1	0
6	0	0	1	1	1	0	1	0	0	0
7	0	1	1	0	0	1	0	0	0	1
8	0	1	0	0	1	0	0	0	1	0
9	1	0	0	0	1	0	0	1	0	0
10	0	1	0	1	0	0	1	0	0	0

Graph Representation II

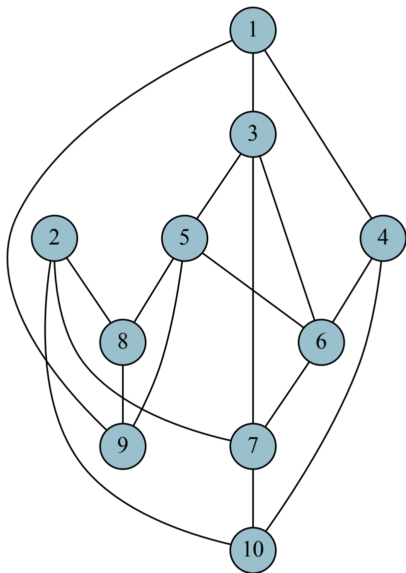
Adjacency Lists

Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

1. For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\text{Adj}(u)$ is the list of edges incident to u.
2. Advantage: space is $O(m + n)$
3. Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$
   3.1 By sorting each list, one can achieve $O(\log n)$ time
   3.2 By hashing "appropriately", one can achieve $O(1)$ time

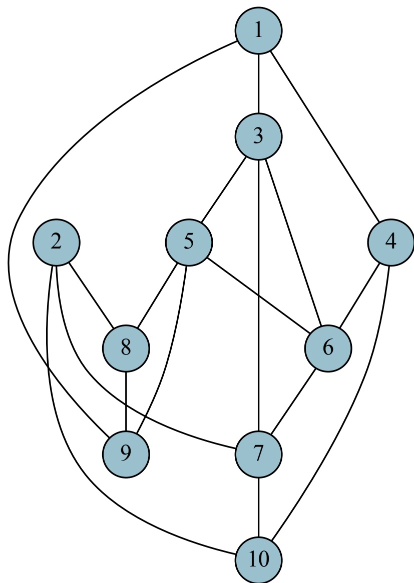
Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

Graph adjacency list example [10 vertices]



vertex	adjacency list
1	3, 4, 9
2	7, 8, 10
3	1, 5, 6, 7
4	1, 6, 10
5	3, 6, 8, 9
6	3, 4, 5, 7
7	2, 3, 6, 10
8	2, 5, 9
9	1, 5, 8
10	2, 4, 7

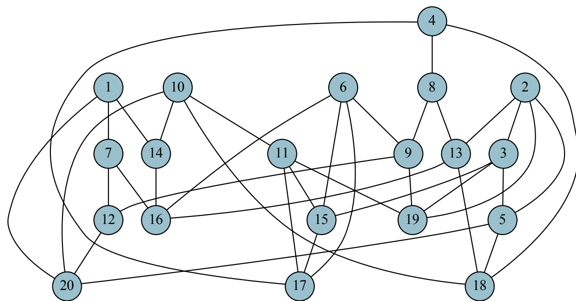
Graph adjacency matrix+list example [10 vertices]



vertex	adjacency list
1	3, 4, 9
2	7, 8, 10
3	1, 5, 6, 7
4	1, 6, 10
5	3, 6, 8, 9
6	3, 4, 5, 7
7	2, 3, 6, 10
8	2, 5, 9
9	1, 5, 8
10	2, 4, 7

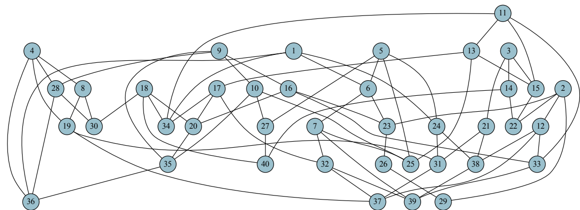
	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	0	0	1	0
2	0	0	0	0	0	0	1	1	0	1
3	1	0	0	0	1	1	1	0	0	0
4	1	0	0	0	0	1	0	0	0	1
5	0	0	1	0	0	1	0	1	1	0
6	0	0	1	1	1	0	1	0	0	0
7	0	1	1	0	0	1	0	0	0	1
8	0	1	0	0	1	0	0	0	1	0
9	1	0	0	0	1	0	0	1	0	0
10	0	1	0	1	0	0	1	0	0	0

Graph adjacency matrix example [20 vertices]



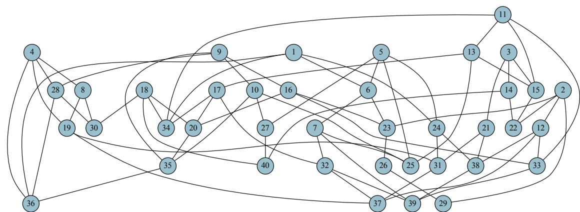
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	2
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1
2	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
3	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0
5	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
8	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1
11	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0
12	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1
13	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0
14	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
15	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0
16	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
17	0	0	0	1	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0
18	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
19	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
20	1	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Graph adjacency matrix example [40 vertices]



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0
6	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
8	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0
14	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
19	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
21	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
24	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
27	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
30	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0
32	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
34	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
36	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
37	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
39	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0

Graph adjacency list example [40 vertices]

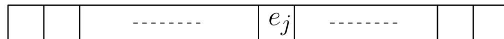


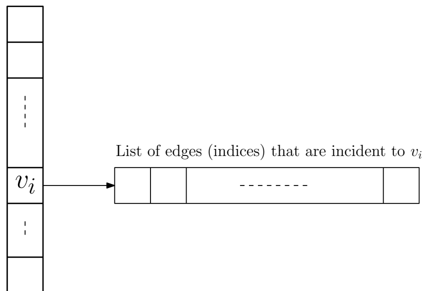
vertex	adjacency list
1	6, 24, 34, 36
2	12, 22, 23, 29
3	14, 15, 21
4	8, 19, 28, 36
5	6, 24, 25, 27
6	1, 5, 7, 23
7	6, 25, 32, 39
8	4, 19, 30
9	10, 16, 28, 35
10	9, 25, 27, 35
11	13, 15, 33, 34
12	2, 33, 37, 38
13	11, 15, 17, 25
14	3, 22, 40
15	3, 11, 13, 22
16	9, 20, 23, 33
17	13, 20, 32, 34
18	20, 30, 34, 40
19	4, 8, 31, 37
20	16, 17, 18, 35
21	3, 31, 38
22	2, 14, 15
23	2, 6, 16, 26
24	1, 5, 31, 38
25	5, 7, 10, 13
26	23, 29
27	5, 10, 40
28	4, 9, 30, 36
29	2, 26
30	8, 18, 28
31	19, 21, 24, 37
32	7, 17, 37, 39
33	11, 12, 16, 39
34	1, 11, 17, 18
35	9, 10, 20, 36
36	1, 4, 28, 35
37	12, 19, 31, 32
38	12, 21, 24, 39
39	7, 32, 33, 38
40	14, 18, 27

A Concrete Representation

- Assume vertices are numbered arbitrarily as $\{1, 2, \ldots, n\}$.
- Edges are numbered arbitrarily as $\{1, 2, \ldots, m\}$.
- Edges stored in an array/list of size m. $E[j]$ is jth edge with info on end points which are integers in range 1 to n.
- Array Adj of size n for adjacency lists. $Adj[i]$ points to adjacency list of vertex i. $Adj[i]$ is a list of edge indices in range 1 to m.

A Concrete Representation





A Concrete Representation: Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multigraphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays: $O(1)$-time operations are easy to understand.
- Can also implement via pointer based lists for certain dynamic graph settings.

16.2
Connectivity

Connectivity

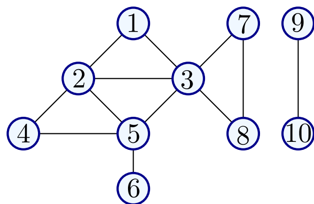
Given a graph $G = (V, E)$:

1. path: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from v_1 to v_k. Note: a single vertex u is a path of length 0.
2. cycle: sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition.
   Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.
3. A vertex u is connected to v if there is a path from u to v.
4. The connected component of u, $\text{con}(u)$, is the set of all vertices connected to u. Is $u \in \text{con}(u)$?

Connectivity contd

Define a relation C on $V \times V$ as uCv if u is connected to v

1. In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
2. Graph is connected if there is only one connected component.



Connectivity Problems

Algorithmic Problems

1. Given graph G and nodes u and v, is u connected to v?
2. Given G and node u, find all nodes that are connected to u.
3. Find all connected components of G.

Can be accomplished in $O(m+n)$ time using **BFS** or **DFS**.
BFS and **DFS** are refinements of a basic search procedure which is good to understand on its own.

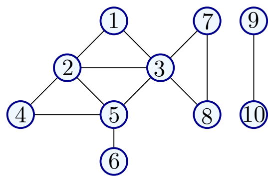
16.3
Computing connected components in undirected graphs using basic graph search

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G, u):
    Visited[1 .. n] ← FALSE
    // ToExplore, S: Lists
    Add u to ToExplore and to S
    Visited[u] ← TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge xy in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
    Output S
```

Example



Properties of Basic Search

Proposition 16.1.

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Once $Visited[i]$ is set to $TRUE$ it never changes. Hence a node is added only once to $ToExplore$. Thus algorithm terminates in at most n iterations of while loop.
- By induction on iterations, can show $v \in S \Rightarrow v \in \text{con}(u)$
- Since each node $v \in S$ was in $ToExplore$ and was explored, no edges in G leave S. Hence no node in $V - S$ is in $\text{con}(u)$.
- Thus $S = \text{con}(u)$ at termination.

Properties of Basic Search

Proposition 16.2.

Explore(G, u) *terminates in* $O(m+n)$ *time.*

Proof: easy exercise

BFS and **DFS** are special case of BasicSearch.

1. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***
2. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***

Search Tree

One can create a natural search tree T rooted at u during search.

```
Explore(G,u):
    array Visited[1..n]
    Initialize: Visited[i] ← FALSE for i = 1,...,n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] ← TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x,y) in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with x as its parent
    Output S
```

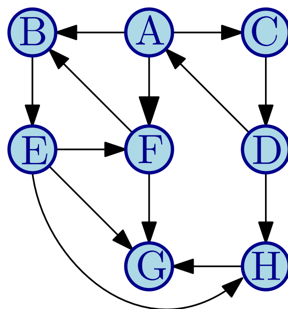
T is a spanning tree of $\text{con}(u)$ rooted at u

Finding all connected components

Exercise: Modify Basic Search to find all connected components of a given graph G in $O(m + n)$ time.

16.4
Directed Graphs and Directed Connectivity

Directed Graphs



Definition 16.1.

A directed graph $G = (V, E)$ consists of

1. set of vertices/nodes V and
2. a set of edges/arcs $E \subseteq V \times V$.

An edge is an ordered pair of vertices. (u, v) different from (v, u).

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

1. Road networks with one-way streets.
2. Web-link graph: vertices are web-pages and there is an edge from page p to page p' if p has a link to p'. Web graphs used by Google with PageRank algorithm to rank pages.
3. Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
4. Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed Graph Representation

Graph $G = (V, E)$ with n vertices and m edges:

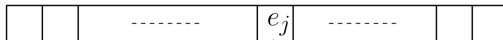
1. **Adjacency Matrix**: $n \times n$ asymmetric matrix A. $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.
2. **Adjacency Lists**: for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

Default representation is adjacency lists.

A Concrete Representation for Directed Graphs

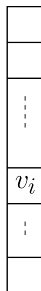
Concrete representation discussed previously for undirected graphs easily extends to directed graphs.

Array of edges E

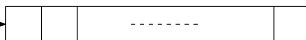


information including end point indices

Array of adjacency lists



List of edges (indices) that are incident to v_i



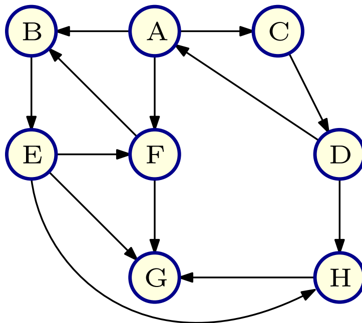
Directed Connectivity

Given a graph $G = (V, E)$:

1. A **(directed) path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k.
   By convention, a single node u is a path of length 0.
2. A **cycle** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$.
   By convention, a single node u is not a cycle.
3. A vertex u can reach v if there is a path from u to v. Alternatively v can be reached from u
4. Let $\mathbf{rch}(u)$ be the set of all vertices reachable from u.

Connectivity contd

Asymmetricity: D can reach B but B cannot reach D



Questions:

1. Is there a notion of connected components?
2. How do we understand connectivity in directed graphs?

16.4.1
Strong connected components

Connectivity and Strong Connected Components

Definition 16.2.

Given a directed graph G, u is **strongly connected** to v if u can reach v and v can reach u. In other words $v \in \mathbf{rch}(u)$ and $u \in \mathbf{rch}(v)$.

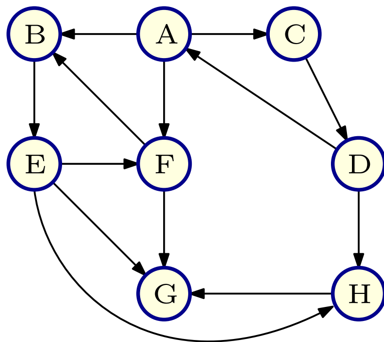
Define relation C where uCv if u is (strongly) connected to v.

Proposition 16.3.

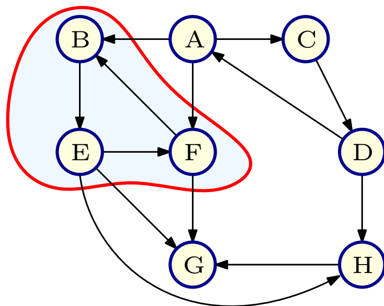
C is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of C: **strong connected components** of G.
They partition the vertices of G.
$\mathrm{SCC}(u)$: strongly connected component containing u.

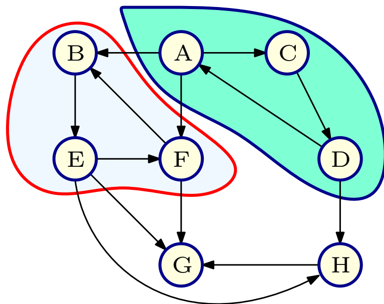
Strongly Connected Components: Example



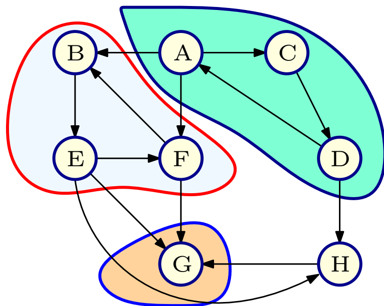
Strongly Connected Components: Example



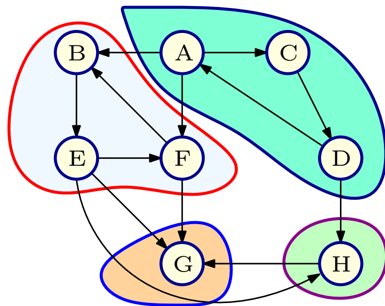
Strongly Connected Components: Example



Strongly Connected Components: Example



Strongly Connected Components: Example



Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\mathbf{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathbf{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute all strongly connected components of G.

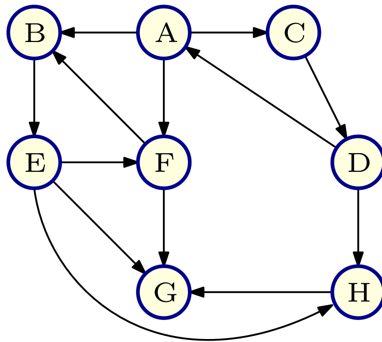
16.4.2
Graph exploration in directed graphs

Basic Graph Search in Directed Graphs

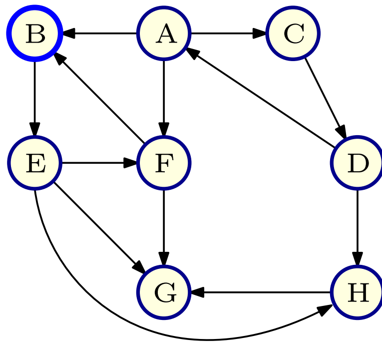
Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

```
Explore(G,u):
    array Visited[1..n]
    Initialize: Set Visited[i] ← FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] ← TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x,y) in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with edge (x,y)
    Output S
```

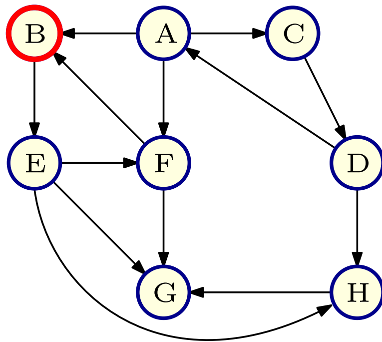
Example



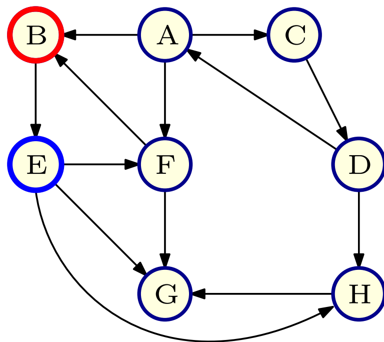
Example



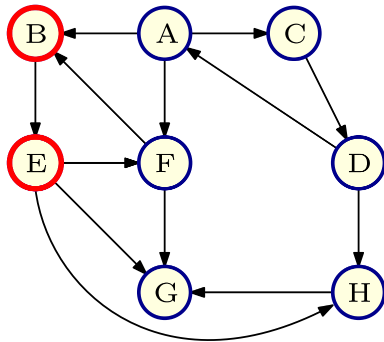
Example



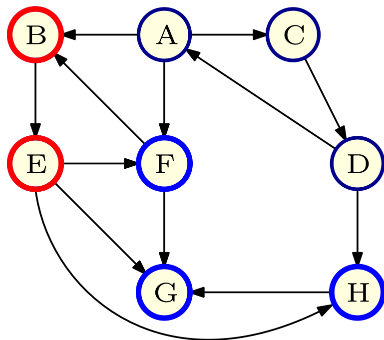
Example



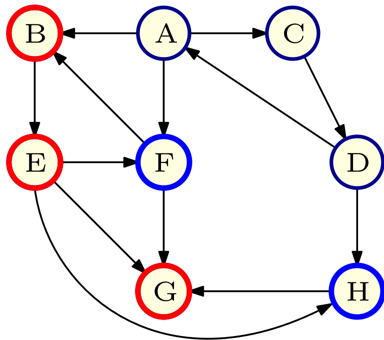
Example



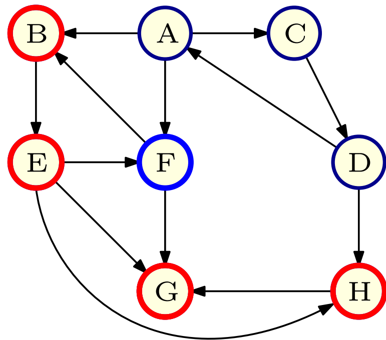
Example



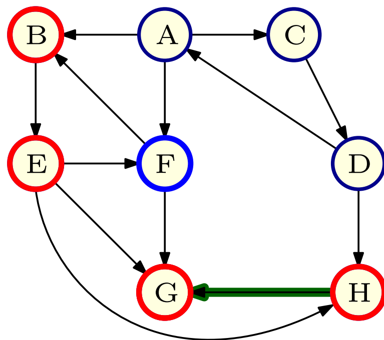
Example



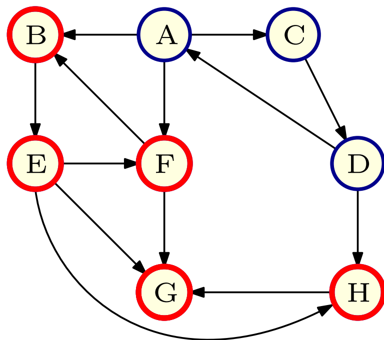
Example



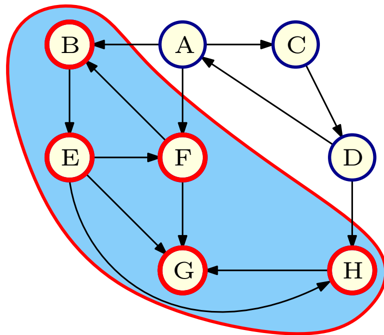
Example



Example



Example



Properties of Basic Search

Proposition 16.4.

$\mathbf{Explore}(G, u)$ *terminates with* $S = \mathbf{rch}(u)$.

Proof Sketch.

- Once $\mathit{Visited}[i]$ is set to TRUE it never changes. Hence a node is added only once to $\mathit{ToExplore}$. Thus algorithm terminates in at most n iterations of while loop.
- By induction on iterations, can show $v \in S \Rightarrow v \in \mathbf{rch}(u)$
- Since each node $v \in S$ was in $\mathit{ToExplore}$ and was explored, no edges in G leave S. Hence no node in $V - S$ is in $\mathbf{rch}(u)$. Caveat: In directed graphs edges can enter S.
- Thus $S = \mathbf{rch}(u)$ at termination.

Properties of Basic Search

Proposition 16.5.

Explore(G, u) *terminates in* $O(m + n)$ *time.*

Proposition 16.6.

T *is a search tree rooted at* u *containing* S *with edges directed away from root to leaves.*

Proof: easy exercises

BFS and **DFS** are special case of Basic Search.

1. Breadth First Search (**BFS**): use queue data structure to implementing the list *ToExplore*
2. Depth First Search (**DFS**): use stack data structure to implement the list *ToExplore*

Exercise

Prove the following:

Proposition 16.7.

Let $S = \mathsf{rch}(u)$. There is no edge $(x, y) \in E$ where $x \in S$ and $y \notin S$.

Describe an example where $\mathsf{rch}(u) \neq V$ and there are edges from $V \setminus \mathsf{rch}(u)$ to $\mathsf{rch}(u)$.

Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\mathrm{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute all strongly connected components of G.

First five problems can be solved in $O(n + m)$ time by via Basic Search (or **BFS**/**DFS**). The last one can also be done in linear time but requires a rather clever **DFS** based algorithm.

16.5
Algorithms via Basic Search

Algorithms via Basic Search - I

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\mathbf{rch}(u)$.

Use $Explore(G, u)$ to compute $\mathbf{rch}(u)$ in $O(n + m)$ time.

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathsf{rch}(v)$. Naive: $O(n(n+m))$

Definition 16.1 (Reverse graph.).

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\mathsf{rch}(u)$ in G^{rev}!

1. **Correctness:** exercise
2. **Running time:** $O(n+m)$ to obtain G^{rev} from G and $O(n+m)$ time to compute $\mathsf{rch}(u)$ via Basic Search. If both $Out(v)$ and $In(v)$ are available at each v then no need to explicitly compute G^{rev}. Can do $Explore(G, u)$ in G^{rev} implicitly.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

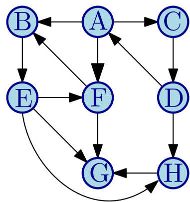
1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathbf{rch}(G, u) \cap \mathbf{rch}(G^{rev}, u)$

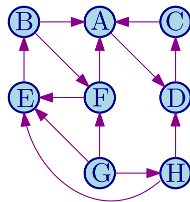
Hence, $\mathrm{SCC}(G, u)$ can be computed with $Explore(G, u)$ and $Explore(G^{rev}, u)$. Total $O(n + m)$ time.

Why can $\mathbf{rch}(G, u) \cap \mathbf{rch}(G^{rev}, u)$ be done in $O(n)$ time?

SCC I: Graph G and its reverse graph G^{rev}



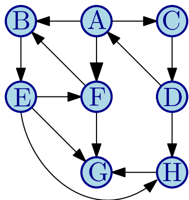
Graph G



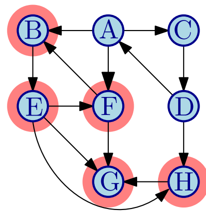
Reverse graph G^{rev}

SCC II: Graph G a vertex **F**

.. and its reachable set **rch**(G, **F**)



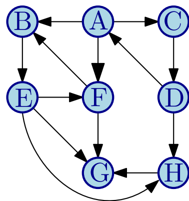
Graph G



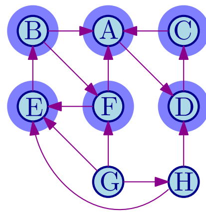
Reachable set of vertices from ***F***

SCC III: Graph G a vertex F

.. and the set of vertices that can reach it in G: **rch(G^{rev}, F)**



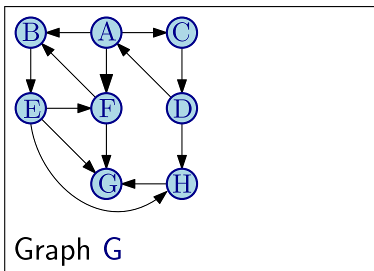
Graph G

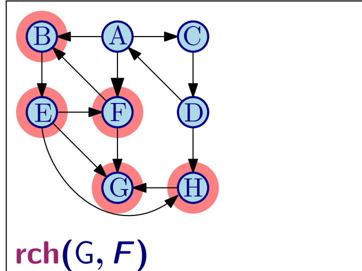


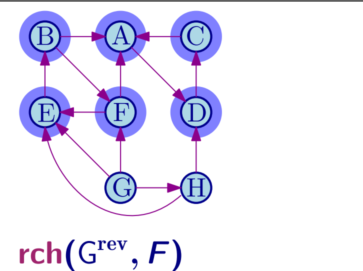
Set of vertices that can reach F, computed via **DFS** in the reverse graph G^{rev}.

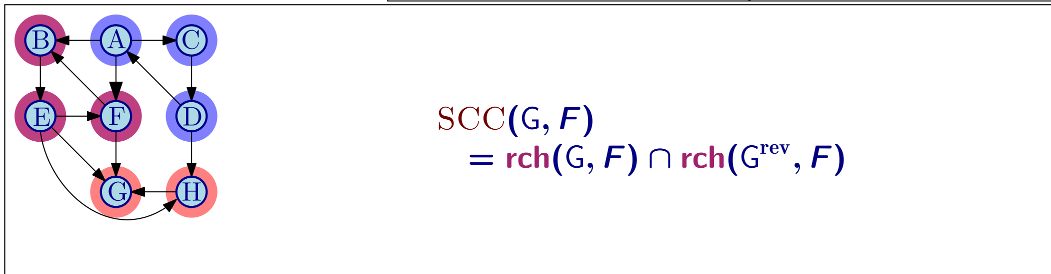
SCC IV: Graph G a vertex **F** and...

its strong connected component in G: $\mathrm{SCC}(\mathsf{G}, \mathbf{F})$









$$\mathrm{SCC}(\mathsf{G}, F) = \mathbf{rch}(\mathsf{G}, F) \cap \mathbf{rch}(\mathsf{G}^{\mathrm{rev}}, F)$$

Algorithms via Basic Search - IV

1. Is G strongly connected?

Pick arbitrary vertex u. Check if $\mathrm{SCC}(G, u) = V$.

Algorithms via Basic Search - V

1. Find <u>all</u> strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Question: Why doesn't removing one strong connected components affect the other strong connected components?

Running time: $O(n(n + m))$.

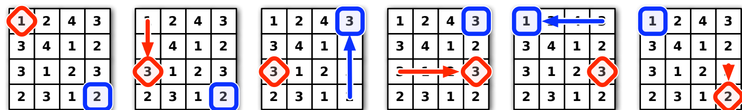
Question: Can we do it in $O(n + m)$ time?

16.6
Exercise: Modefling problems using graphs

Modeling Problems as Search

The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.



A five-move solution for a 4×4 Vidrach Itky Leda puzzle.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle above, your algorithm would return the number 5.

Undirected vs Directed Connectivity

Consider following problem.

- Given undirected graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes). R and B non-empty.
- Describe linear-time algorithm to decide whether every red node can reach every blue node.

How does the problem differ in directed graphs?

Undirected vs Directed Connectivity

Consider following problem.

- Given directed graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes).
- Describe linear-time algorithm to decide whether every red node can be reached by some blue node.