Pre-lecture brain teaser

Remembering the edit distance example we saw in class last time, we formaluted the processing of the recursion as a table:

	ε	D	R	E	A	D
ε						
D						
E						
E						
D						

Is there an easier way to get the minimum alignment without having to calculate all the values in the cell?

ECE-374-B: Lecture 14 - Graph search

Instructor: Nickvash Kani

October 16, 2025

University of Illinois Urbana-Champaign

Remembering the edit distance example we saw in class last time, we formaluted the processing of the recursion as a table:

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1	0				
E	2					
E	3					
D	4					

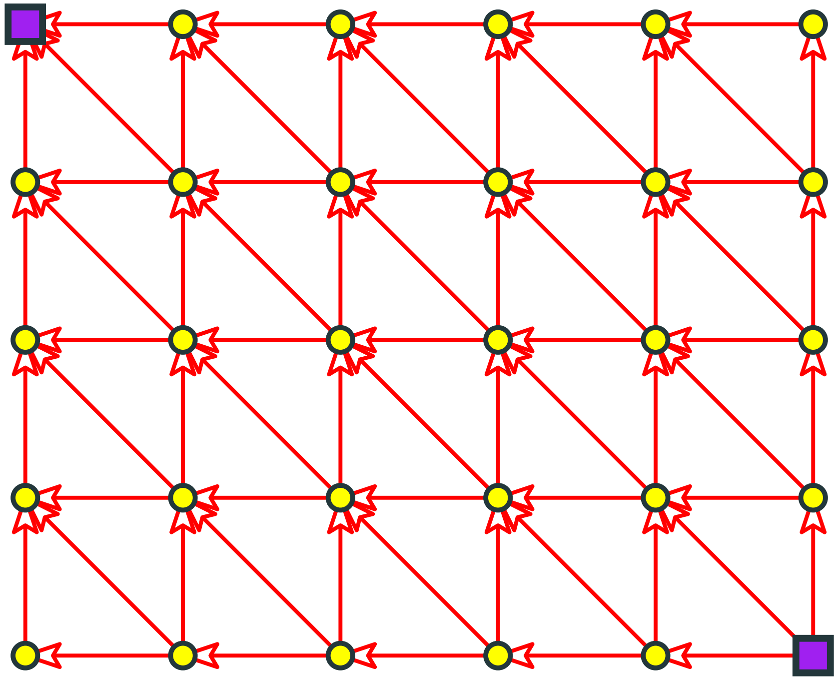
Alignment the last letters of both sequences on top of each other

Is there an easier way to get the minimum alignment without having to calculate all the values in the cell?

Pre-lecture brain teaser

Remembering the edit distance example we saw in class last time, we formaluted the processing of the recursion as a table:

	ε	D	R	E	A	D
ε						
D						
E						
E						
D						

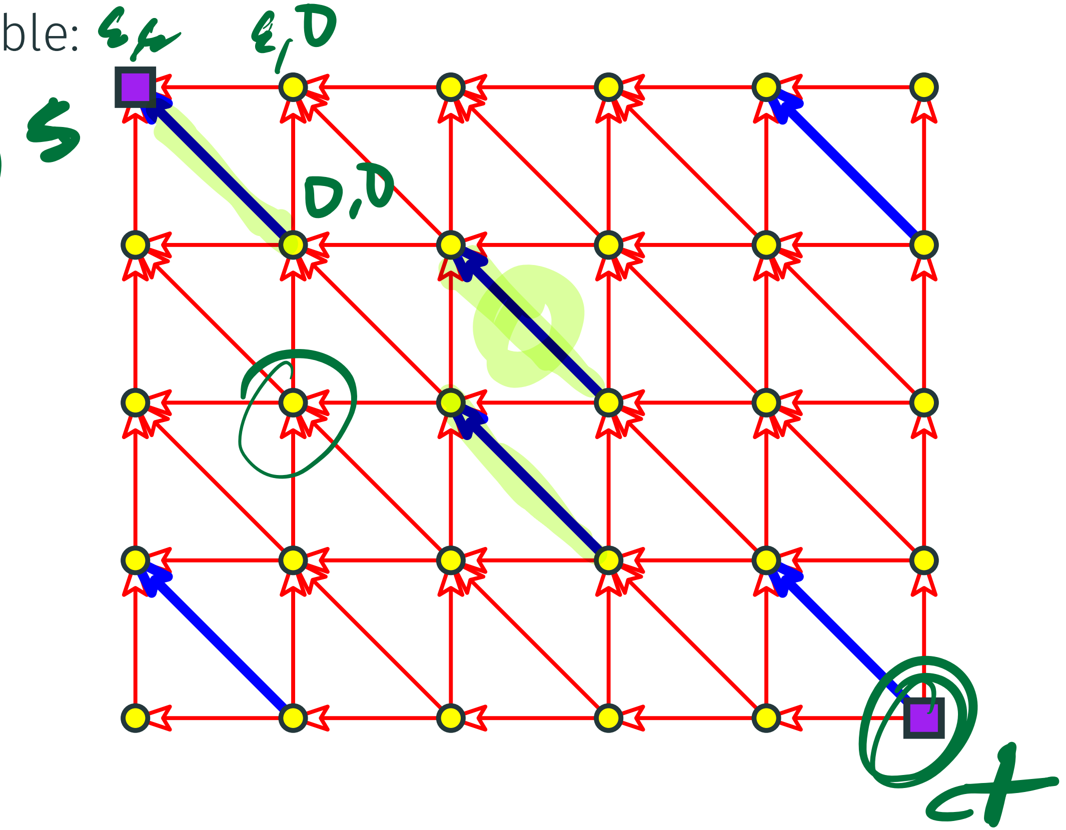


Look at the flow of the computation!

Pre-lecture brain teaser

Remembering the edit distance example we saw in class last time, we formaluted the processing of the recursion as a table:

	ε	D	R	E	A	D
ε						
D						
E		0				
E						
D						



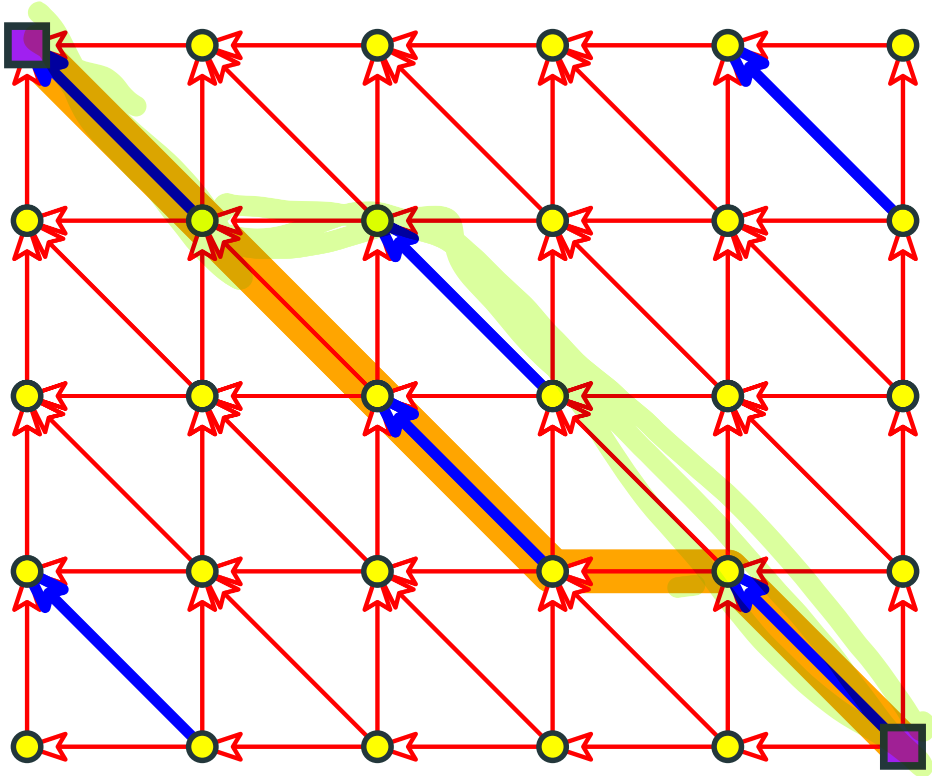
Look at the flow of the computation!

Pre-lecture brain teaser

Remembering the edit distance example we saw in class last time, we formaluted the processing of the recursion as a table:

2 DREAD
DEED

	ε	D	R	E	A	D
ε						
D						
E						
E						
D						



Graph Basics

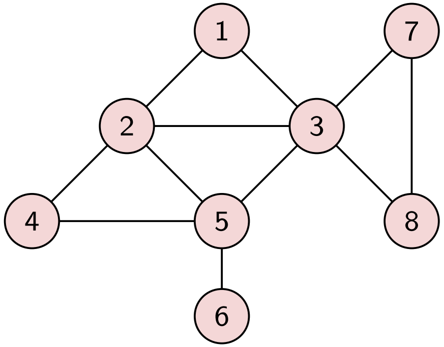
Why Graphs?

- 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.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics

Graph

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

- V is a set of vertices (also referred to as nodes/points)
- 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 $u \neq v$.



Example

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

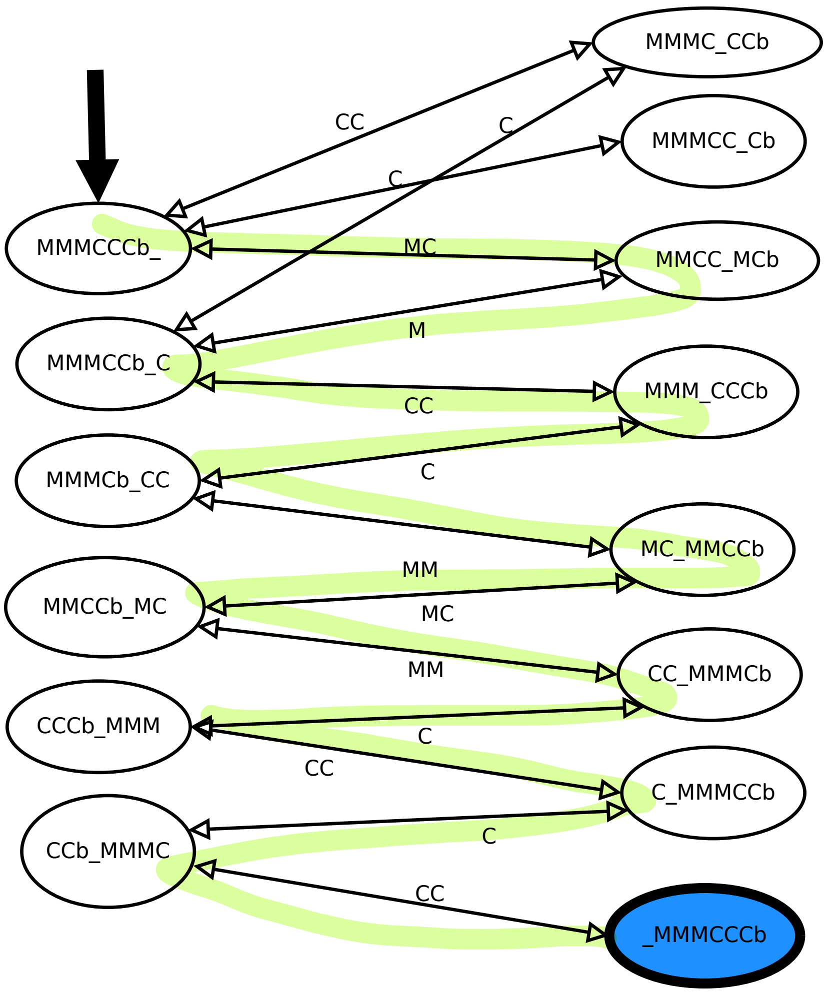
- Three missionaries, three cannibals, one boat, one river
- Boat carries two people, must have at least one person
- Must all get across
- At no time can cannibals outnumber missionaries

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.

Lions and buffalo

All bad names to a simple problem...

Problems on DFAs and NFAs sometimes are just problems on graphs

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

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.

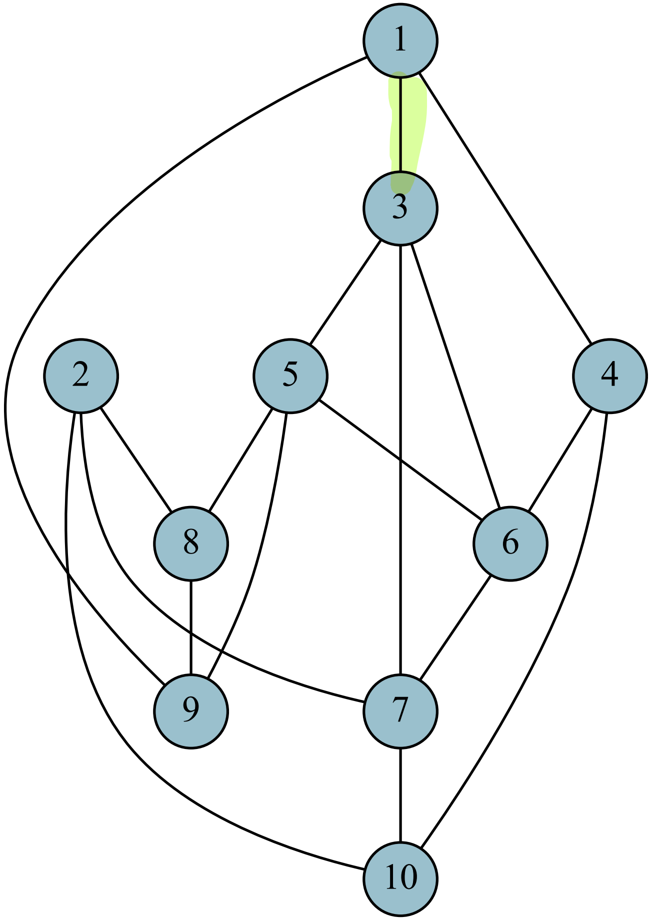
- u and v are the end points of an edge $\{u, v\}$
- Multi-graphs allow
    - loops which are edges with the same node appearing as both end points
    - multi-edges: different edges between same pairs of nodes
- In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

Graph Representation I

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

- $A[i, j] = A[j, i] = 1$ if $\{i, j\} \in E$ and $A[i, j] = A[j, i] = 0$ if $\{i, j\} \notin E$.
- Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
- 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

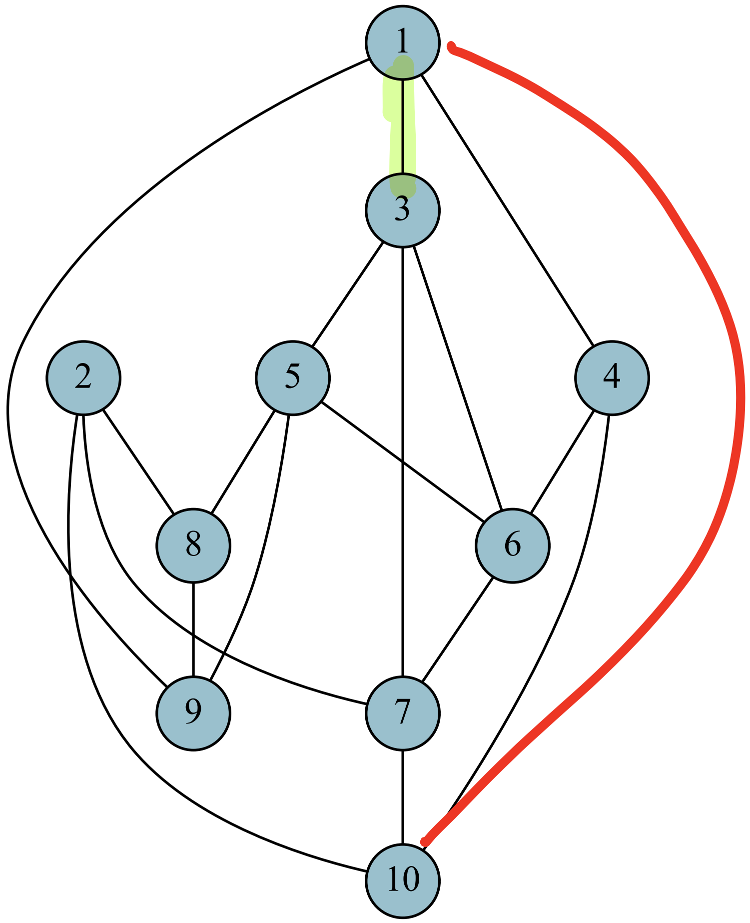
Adjacency Lists

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

- 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.
- Advantage: space is $O(m + n)$
- Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$
  - By sorting each list, one can achieve $O(\log n)$ time
  - By hashing "appropriately", one can achieve $O(1)$ time

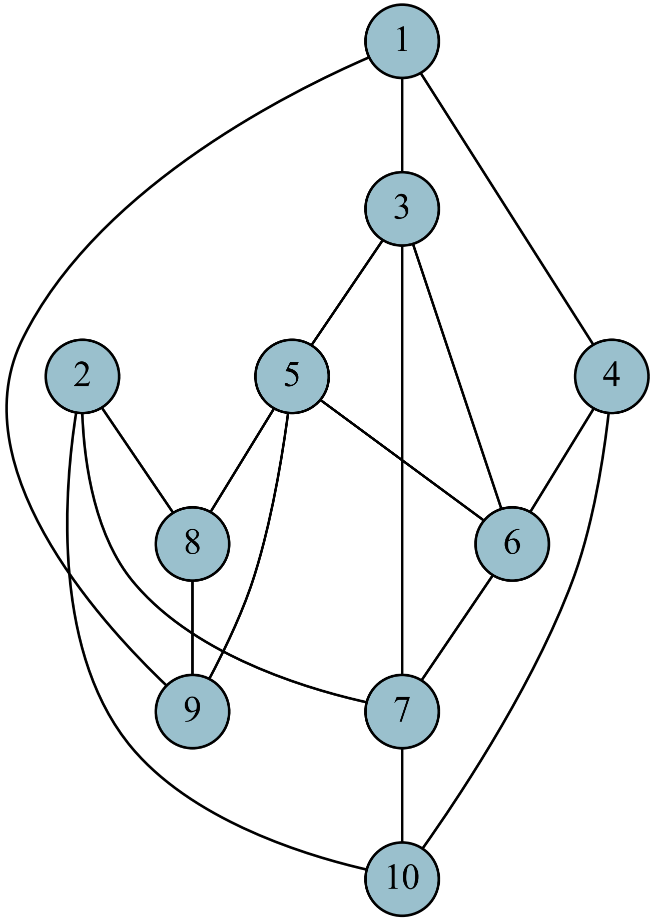
Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

Graph adjacency list example [10 vertices]



vertex	adjacency list
1	3, 4, 9 10
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

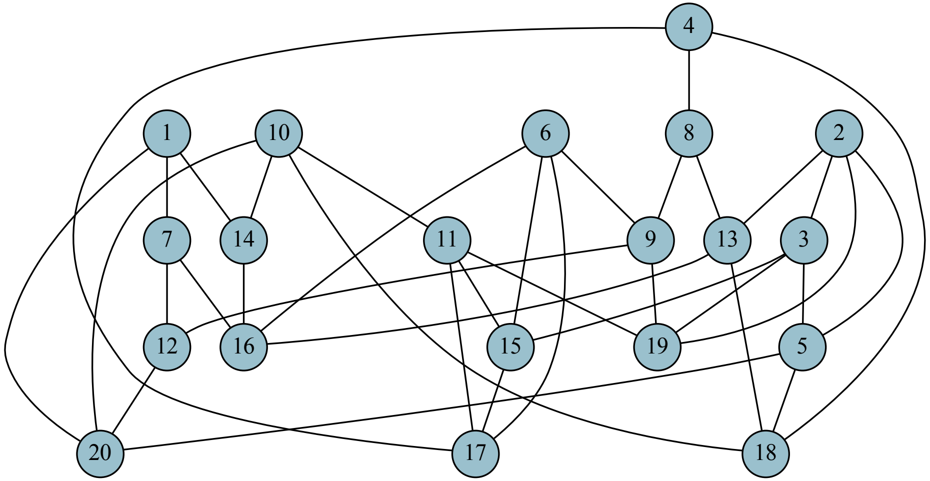
Graph adjacency matrix+list example [10 vertices]



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

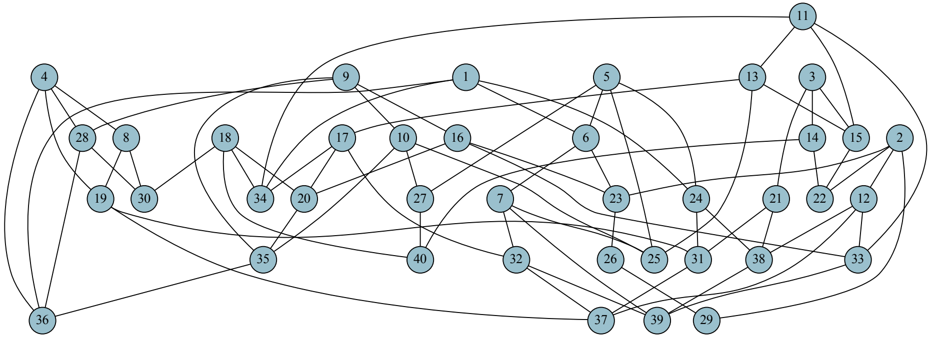
	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	0	0	1	0
2	0	0	0	0	0	0	1	1	0	1
3	1	0	0	0	1	1	1	0	0	0
4	1	0	0	0	0	1	0	0	0	1
5	0	0	1	0	0	1	0	1	1	0
6	0	0	1	1	1	0	1	0	0	0
7	0	1	1	0	0	1	0	0	0	1
8	0	1	0	0	1	0	0	0	1	0
9	1	0	0	0	1	0	0	1	0	0
10	0	1	0	1	0	0	1	0	0	0

Graph adjacency matrix example [20 vertices]



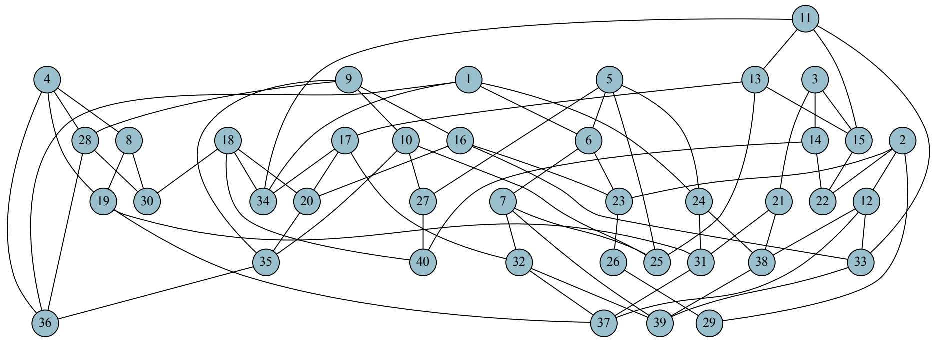
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1
2	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
3	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0
5	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0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

Graph adjacency list example [40 vertices]

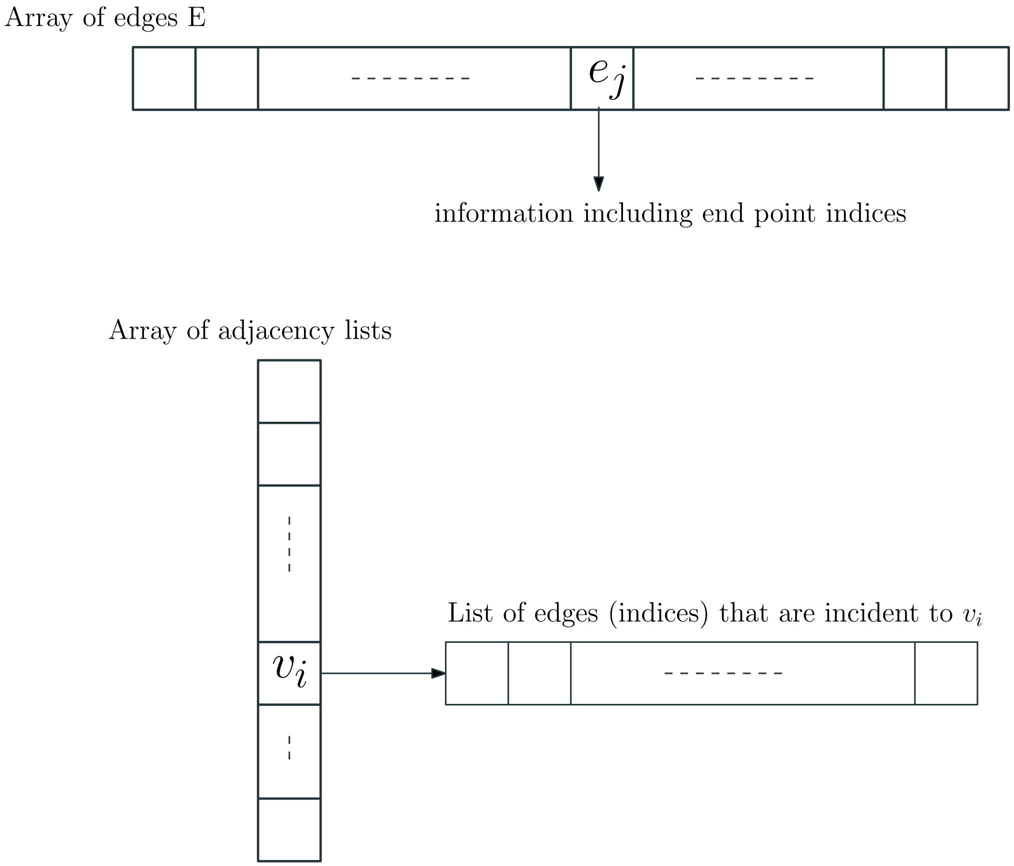


vertex	adjacency list
1	6, 24, 34, 36
2	12, 22, 23, 29
3	14, 15, 21
4	8, 19, 28, 36
5	6, 24, 25, 27
6	1, 5, 7, 23
7	6, 25, 32, 39
8	4, 19, 30
9	10, 16, 28, 35
10	9, 25, 27, 35
11	13, 15, 33, 34
12	2, 33, 37, 38
13	11, 15, 17, 25
14	3, 22, 40
15	3, 11, 13, 22
16	9, 20, 23, 33
17	13, 20, 32, 34
18	20,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

Connectivity I

Connectivity

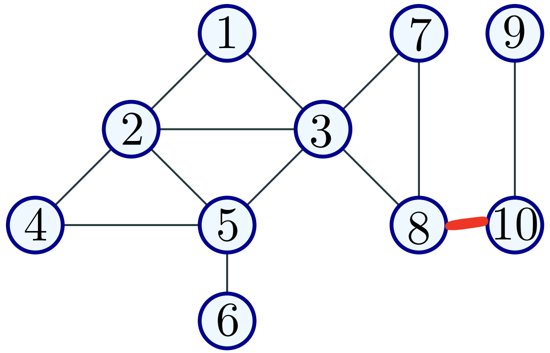
Given a graph $G = (V, E)$:

- 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.
- 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.
- A vertex u is connected to v if there is a path from u to v.
- The connected component of u, $\text{con}(u)$, is the set of all vertices connected to u. Is $u \in \text{con}(u)$? Yes

Connectivity II

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

- In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
- Graph is <u>connected</u> if there is only one connected component.



Connectivity Problems

Algorithmic Problems

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

Connectivity Problems

Algorithmic Problems

- Given graph G and nodes u and v, is u connected to v?
- Given G and node u, find all nodes that are connected to u.
- 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.

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 $n = |V|$.

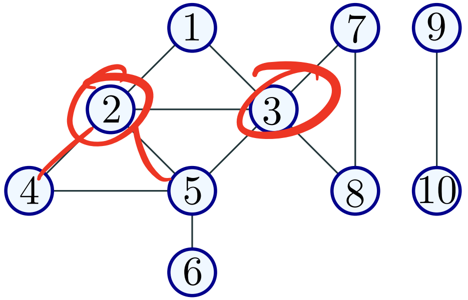
```
Explore(G,u):
    Visited[1..n] ← FALSE
    // ToExplore, S: Lists
    Add u to ToExplore and to S
    Visited[u] ← TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge xy in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y]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

u

Example

```
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
```

G:



Explore (G, 1)

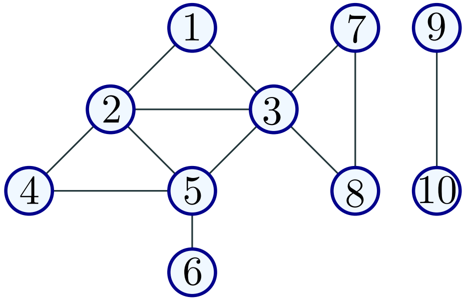
ToExplore	S
~~1~~	1
~~2~~	2
~~3~~	3
4	4
5 8	~~5~~ 8

Example

```
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
```

$n = |V|$

$m = |E|$



con(u)

while Visited != All true, then push unvisited node onto ToExplore

Running Time: $O(n+m)$

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
```

Finding all connected components

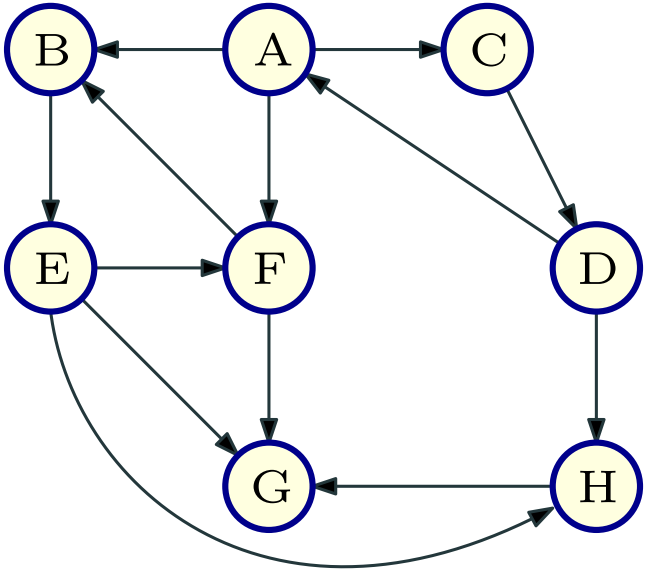
Modify Basic Search to find all connected components of a given graph G in $O(m + n)$ time.

Directed Graphs and Directed Connectivity

Directed Graphs

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

- set of vertices/nodes V and
- 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:

- Road networks with one-way streets.
- 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.
- Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
- Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed Graph Representation

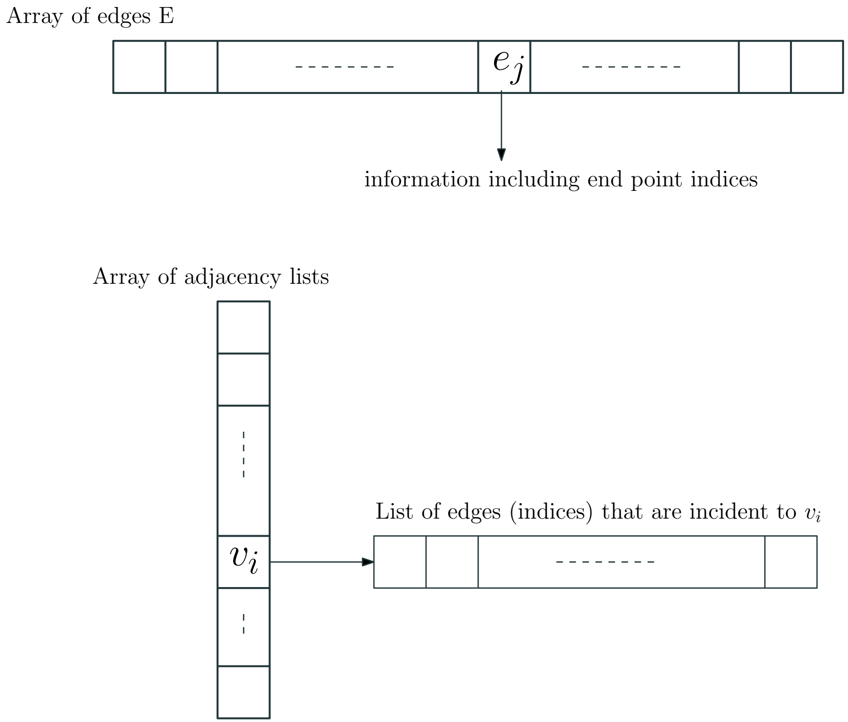
Graph $G = (V, E)$ with n vertices and m edges:

- 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 $A[v, u]$.
- 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.



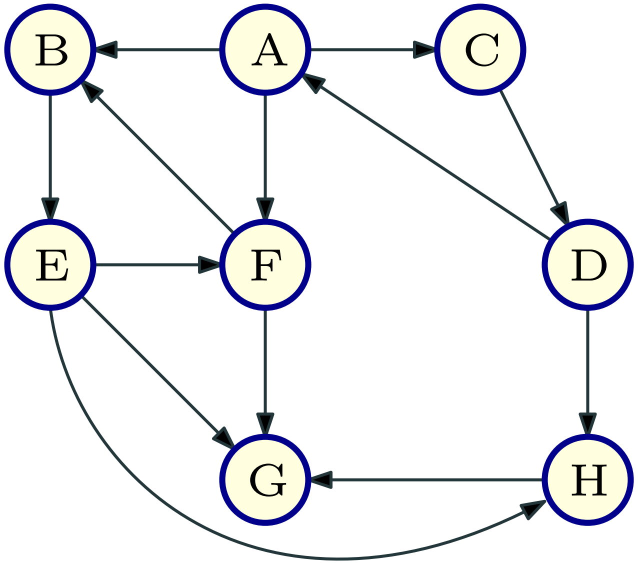
Directed Connectivity

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

- 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 $1 \le i \le k-1$. 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.
- 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 $1 \le i \le k-1$ and $(v_k, v_1) \in E$.
  By convention, a single node u is not a cycle.
- A vertex u can reach v if there is a path from u to v. Alternatively v can be reached from u
- Let $\mathrm{rch}(u)$ be the set of all vertices reachable from u.

Directed Connectivity II

Asymmetricity: *D* can reach *B* but *B* cannot reach *D*



Questions:

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

Strong connected components

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

Connectivity and Strong Connected Components

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

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

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

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

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

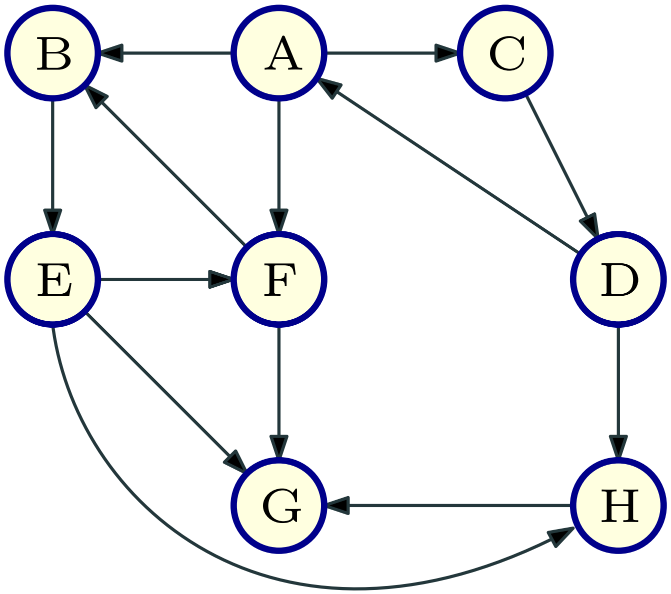
Definition
Given a directed graph G, u is strongly connected to v if u can reach v and v can reach u. In other words $v \in \mathsf{rch}(u)$ and $u \in \mathsf{rch}(v)$.

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

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

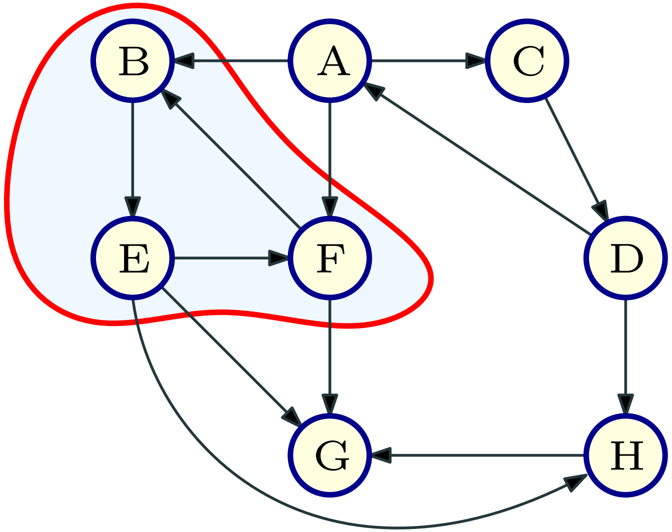
Equivalence classes of C: strong connected components of G.
They partition the vertices of G.
$SCC(u)$: strongly connected component containing u.

Strongly Connected Components: Example

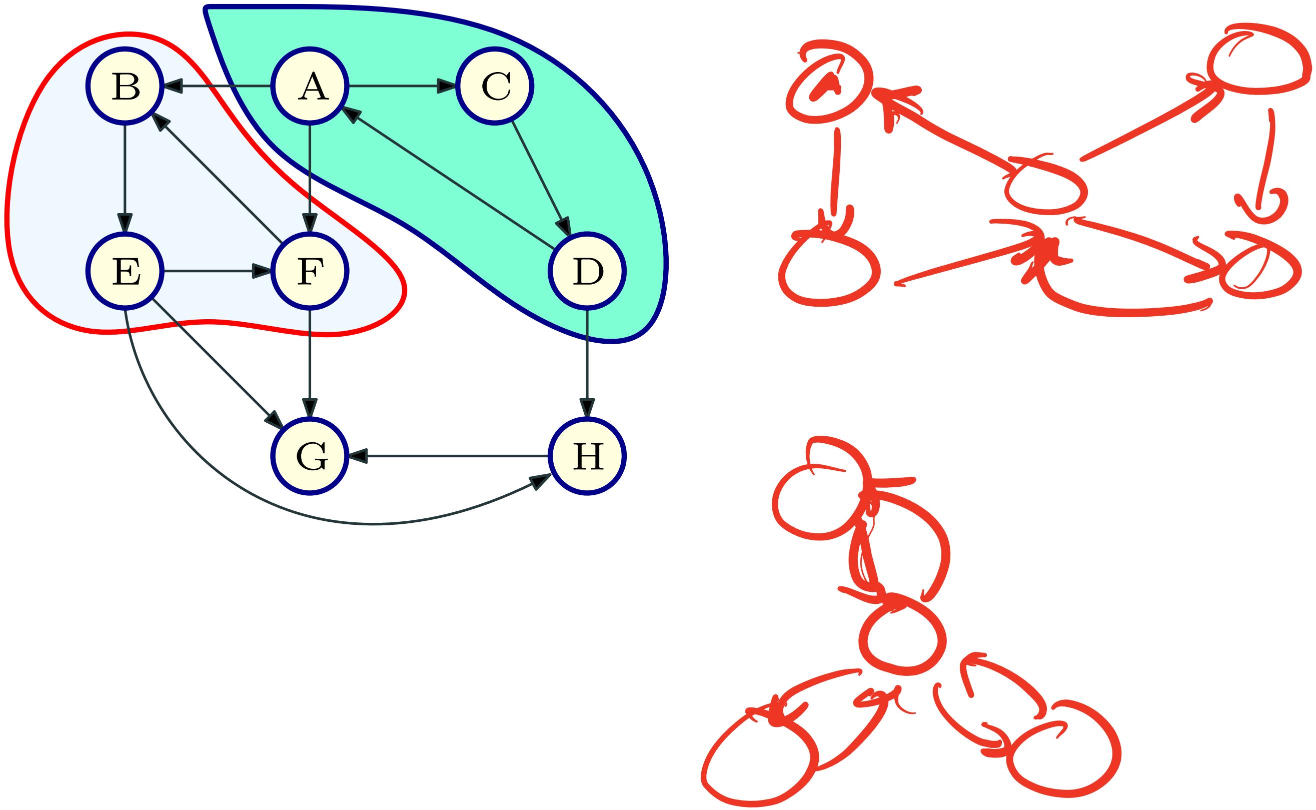


Strongly Connected Components: Example

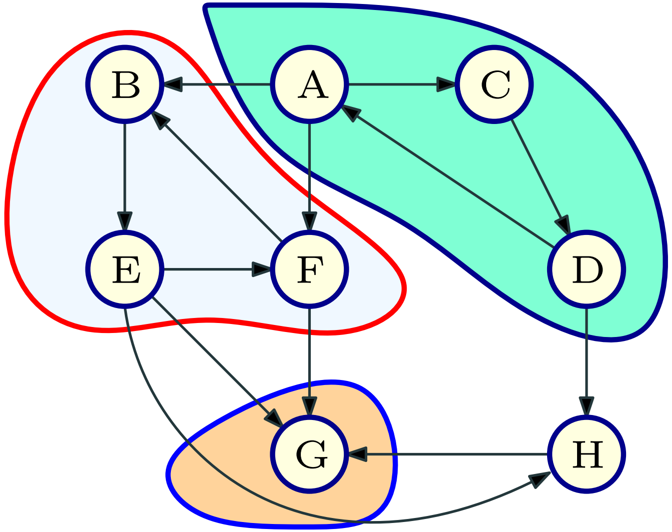
scc(B)
scc(E)
scc(F)



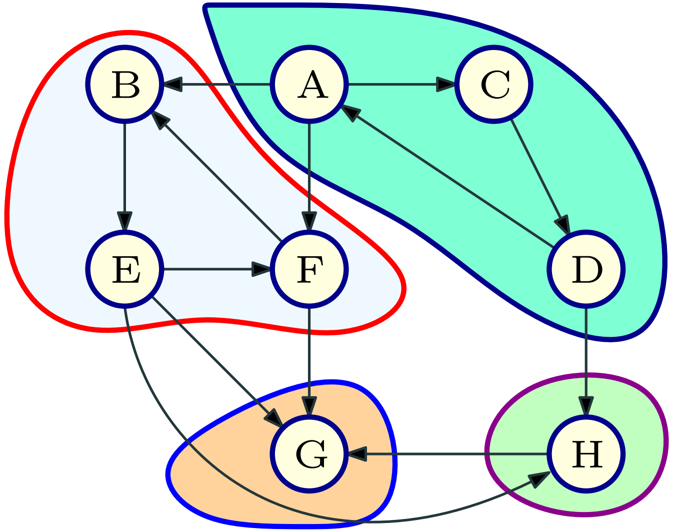
Strongly Connected Components: Example



Strongly Connected Components: Example



Strongly Connected Components: Example



Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\mathsf{rch}(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in \mathsf{rch}(v)$.
- Find the strongly connected component containing node u, that is $SCC(u)$.
- Is G strongly connected (a single strong component)?
- Compute <u>all</u> strongly connected components of G.

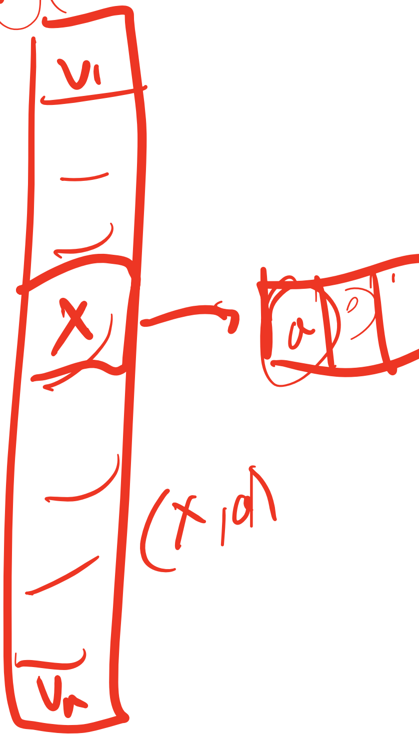
Graph exploration in directed graphs

Basic Graph Search in Directed Graphs

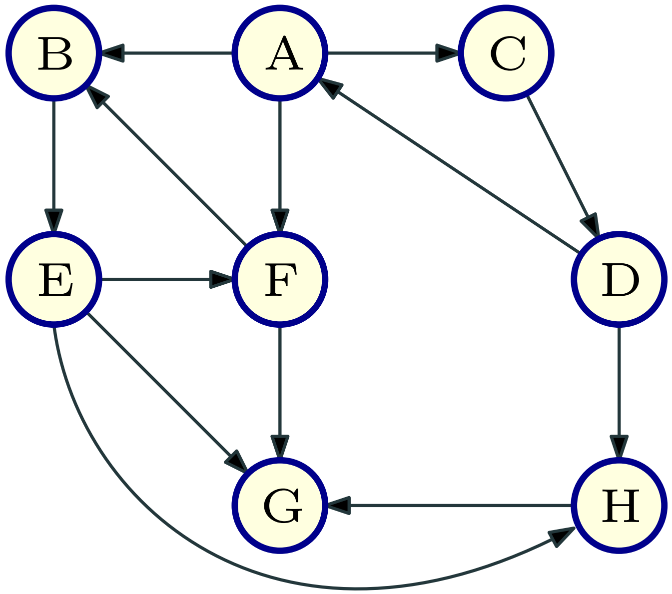
Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

$O(n+m)$

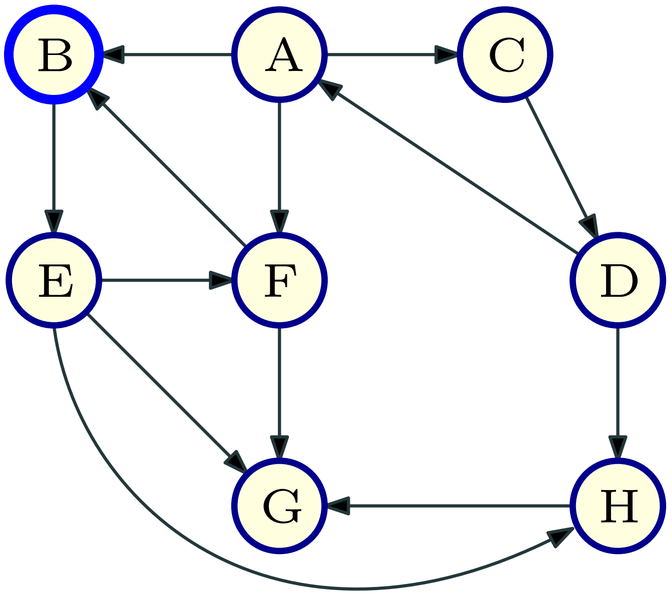
```
Explore(G,u):
    array Visited[1..n]
    Initialize: Set Visited[i] ← FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] ← TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x,y)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



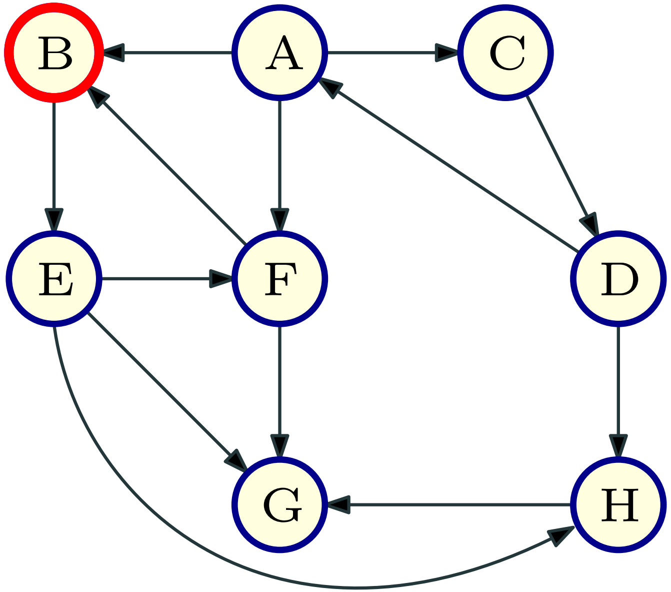
Example



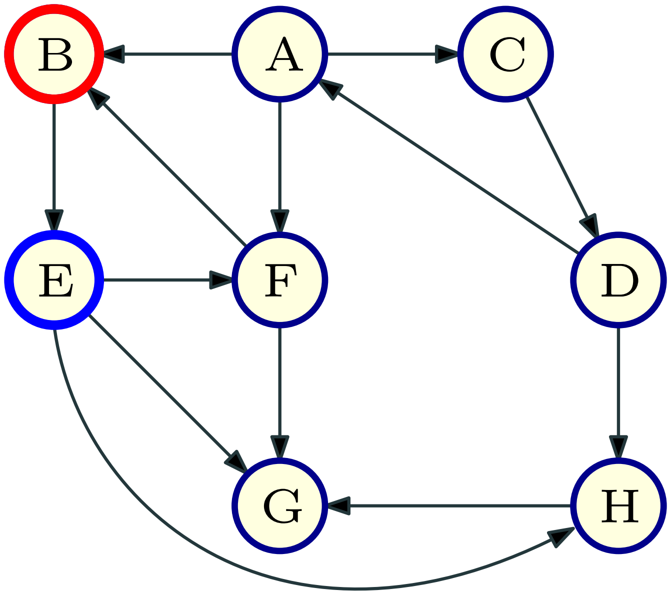
Example



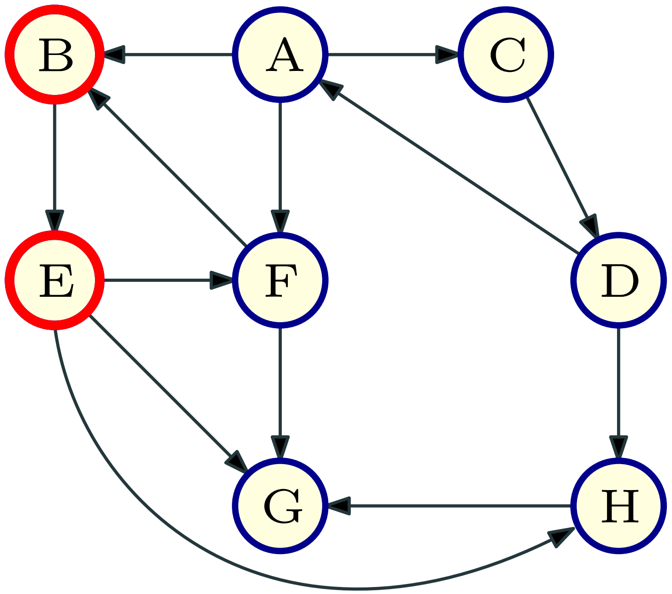
Example



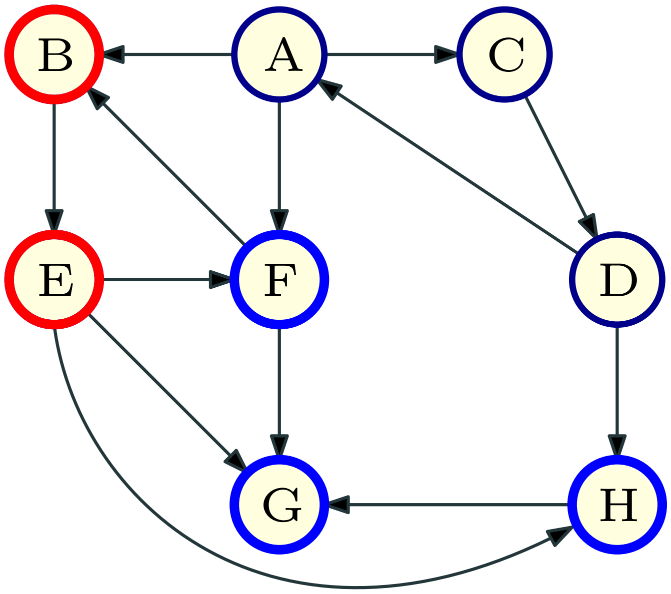
Example



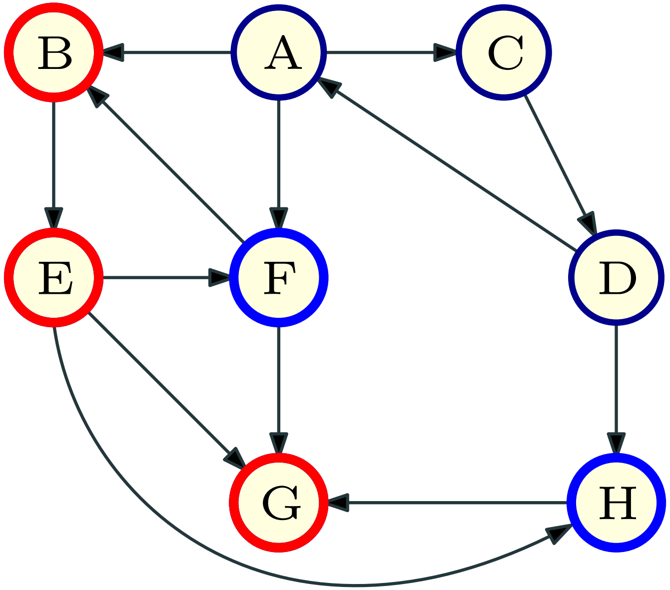
Example



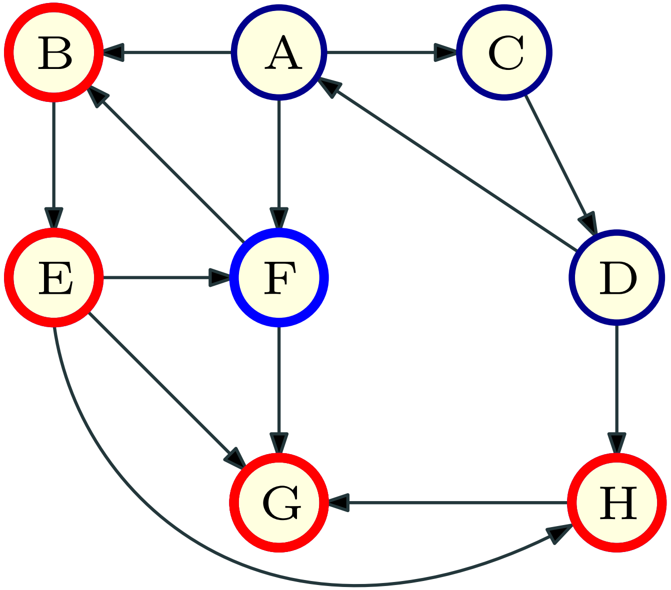
Example



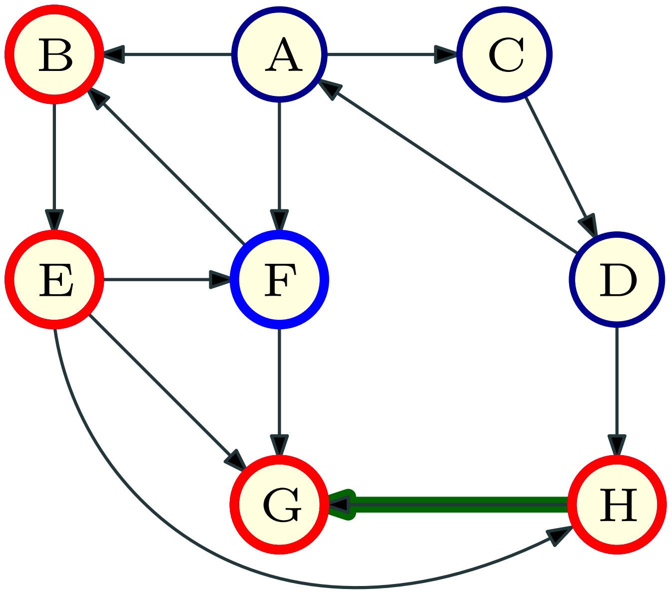
Example



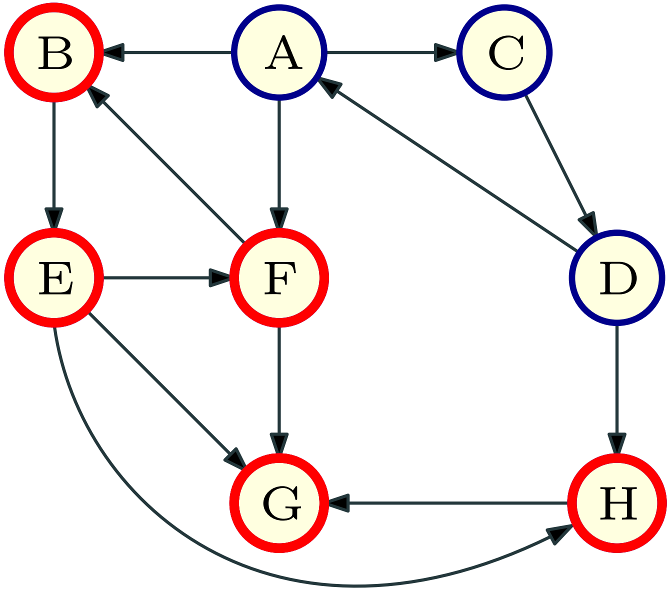
Example



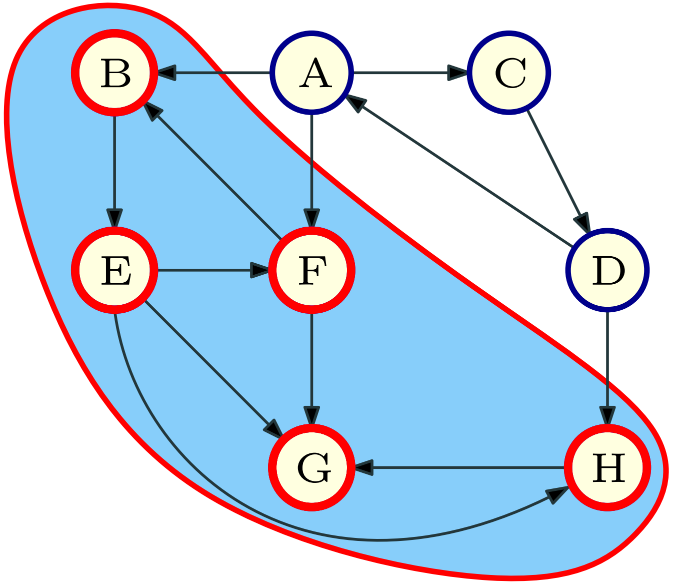
Example



Example



Example



Properties of Basic Search

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

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 $v \in S \Rightarrow v \in \text{rch}(u)$
- 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 $\text{rch}(u)$. Caveat: In directed graphs edges can enter S.
- Thus $S = \text{rch}(u)$ at termination.

$\square$

Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v? Explore, return if v in S.
- Given G and u, compute $\mathrm{rch}(u)$. Explore return S
- Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$. Explore(G^{rev}, u)
- Find the strongly connected component containing node u, that is $SCC(u)$.
- Is G strongly connected (a single strong component)?
- Compute <u>all</u> strongly connected components of G.

$\to$ G^{rev} = G with all the edges reversed

Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\mathsf{rch}(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in \mathsf{rch}(v)$.
- Find the strongly connected component containing node u, that is $SCC(u)$.
- Is G strongly connected (a single strong component)?
- Compute <u>all</u> 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 (next lecture).

Algorithms via Basic Search

Algorithms via Basic Search - I

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\mathsf{rch}(u)$.

Algorithms via Basic Search - I

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\text{rch}(u)$.

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

Algorithms via Basic Search - II

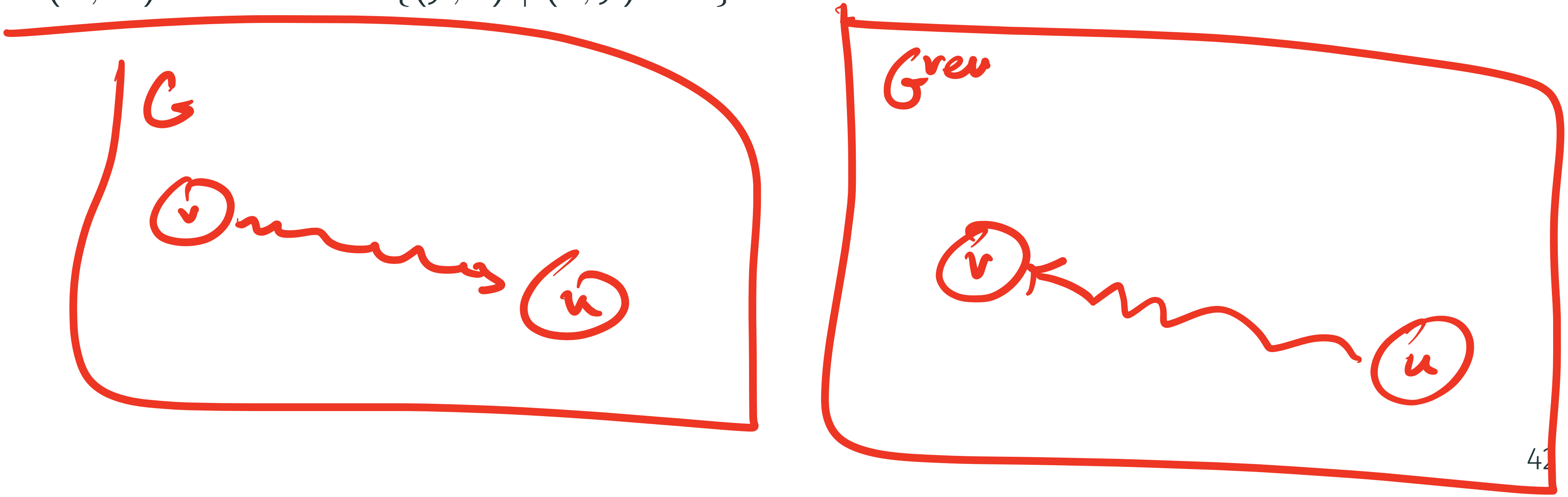
- Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

Algorithms via Basic Search - II

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

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

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



Algorithms via Basic Search - II

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

Definition (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 $\text{rch}(u)$ in G^{rev}!

- **Running time:** $O(n+m)$ to obtain G^{rev} from G and $O(n+m)$ time to compute $\text{rch}(u)$ 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.

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

$Explore(G, u) = rch(u)$

$Explore(G^{rev}, u) = \{v\} \mid rch(v) = u)$

$SCC(u)$

Algorithms via Basic Search - III

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

- Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

Algorithms via Basic Search - III

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

- Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

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

Algorithms via Basic Search - III

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

- Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

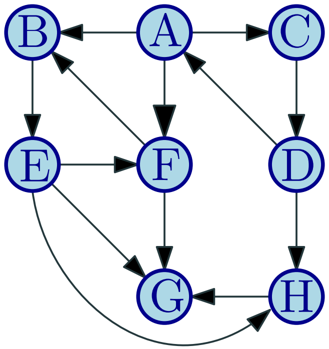
$SCC(G, u) = \mathbf{rch}(G, u) \cap \mathbf{rch}(G^{rev}, u)$

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

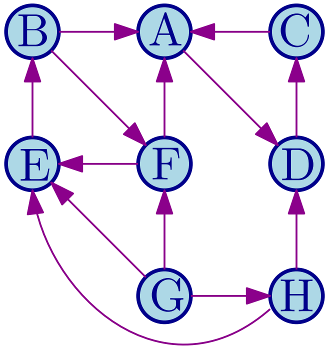
Why can $\mathbf{rch}(G, u) \cap \mathbf{rch}(G^{rev}, u)$ be done in $O(n)$ time?

SCC I

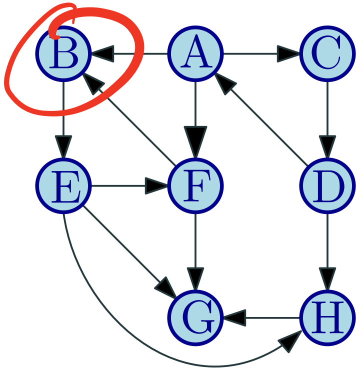
Graph G and its reverse graph G^{rev}



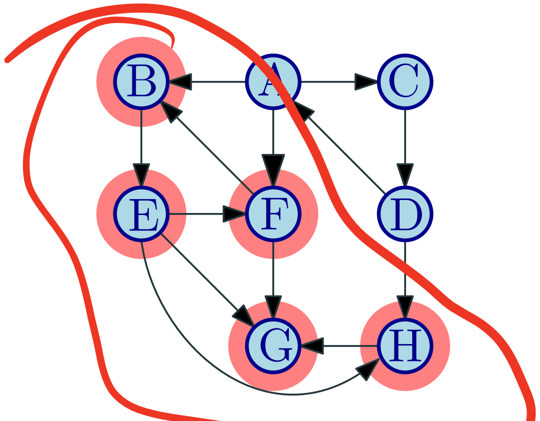
Graph G



Graph G a vertex F and its reachable set (G, F)

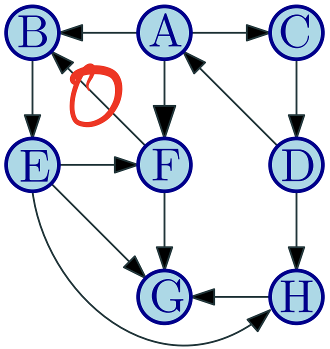


Graph G

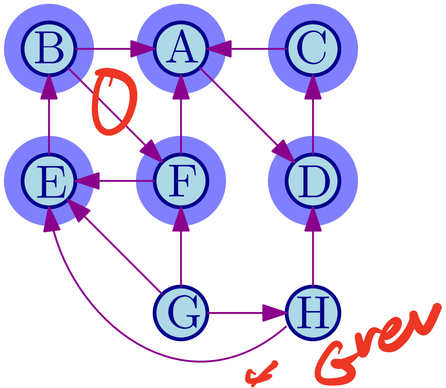


SCC III

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

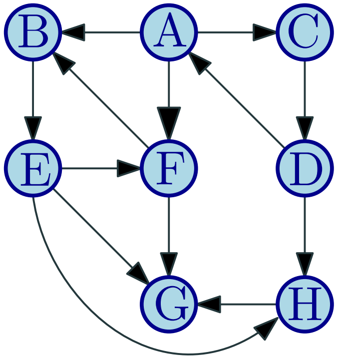


Graph G

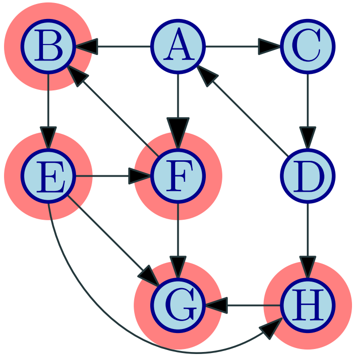


SCC IV: ...

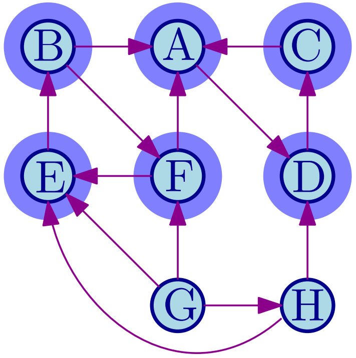
Graph G a vertex F and its strong connected component in G: SCC(G, F)



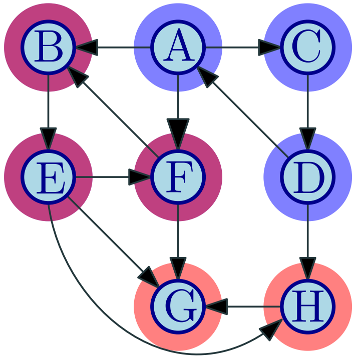
Graph G



rch(G, F)



rch(G^{rev}, F)



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

Algorithms via Basic Search - IV

- Is G strongly connected?

- Is G strongly connected?

Pick arbitrary vertex u. Check if $SCC(G, u) = V$.

Algorithms via Basic Search - V

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

- 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
```

Algorithms via Basic Search - V

- 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?

Algorithms via Basic Search - V

- 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: $O(n(n+m))$.

Algorithms via Basic Search - V

- 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?

Find out next time.....