# Algorithms for Data Processing
# Lecture III: Graph Algorithms – Undirected Graphs

**Alessandro Artale**

Free University of Bozen-Bolzano
Faculty of Computer Science
http://www.inf.unibz.it/˜artale
artale@inf.unibz.it

2019/20 – First Semester
MSc in Computational Data Science — UNIBZ

# Graph Representation: Size of the Input

- A Graph $G = (V, E)$ has two natural parametres:
  - Number of nodes. $n = |V|$;
  - Number of edges. $m = |E|$.
- Running time/Space required will be given in terms of both of these two parameters.

# Graph Representation: Adjacency Matrix

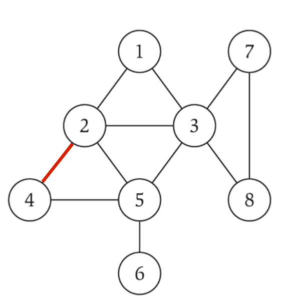
Adjacency Matrix. For a graph $G$ with $n$ vertices, is a $n \times n$ matrix with $A[u, v] = 1$ if $(u, v)$ is an edge.

- Each edge is mentioned twice in the matrix when $G$ is undirected, i.e., the matrix is symmetric.

Properties:

1. Search/Delete. Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
2. Storage. Space required is $\Theta(n^2)$—when the $G$ has many fewer edges more compact representations are possible.
3. They are not efficient to check all incident edges which takes $\Theta(n)$ time.

# Graph Representation: Adjacency Matrix




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

# Graph Representation: Adjacency List

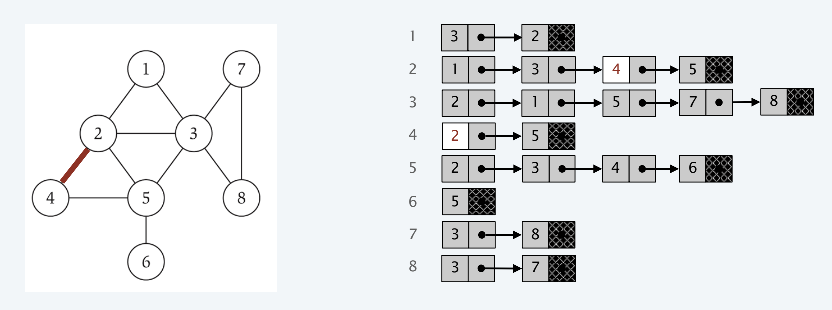
Adjacency list. Vertex-indexed array of lists.

- The array **Adj** when indexed with a vertex $v$, $\textbf{Adj}[v]$, is a pointer to the list of all vertices adjacent to $v$.
- Each edge is mentioned twice (when $G$ is undirected).

Properties:

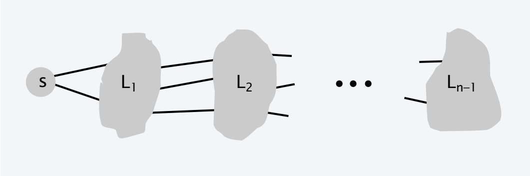
1. Search/Delete. Checking if $(u, v)$ is an edge takes $\Theta(\textbf{deg}(u))$ time.
2. Storage. Space is $\Theta(m + n)$: Since each edge appears twice, and $2 \cdot m \in O(m)$, and we need an array of $n$ pointers to initialize **Adj**.
   - **Note.** Since $m \leq n^2$, $\Theta(m + n)$ is $O(n^2)$, i.e., much better when $G$ is *sparse*.
3. Identifying all incident edges to $v$ takes $\Theta(\textbf{deg}(v))$ time better than $\Theta(n)$.

# Graph Representation: Adjacency List



# Breadth-First Search (BFS)

- *s*-*t* connectivity problem (Reachability). Given two nodes, $s, t$, is there a path between $s$ and $t$?
- BFS intuition. Explore outward from the vertex $s$ in all possible directions, adding vertices one *layer* at a time.



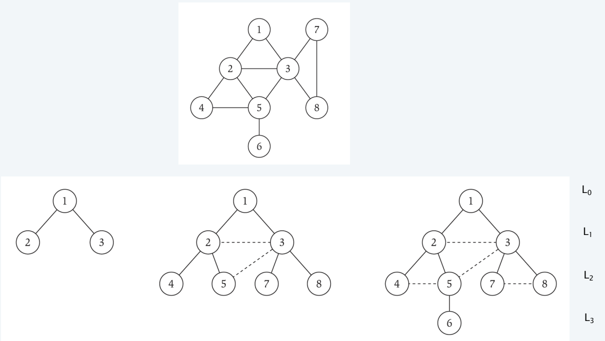

Layers $L_1, , L2, L_3, \ldots$ are constructed in the following way:

1. $L_1$ consists of all vertices adjacent to $s$;
2. $L_{j+1}$ consists of all vertices that: *i*) Do not belong to an earlier layer, and *ii*) Are adjacent to a vertex in layer $L_j$.

# BFS — Spanning Tree

- BFS traverses a connected component of an undirected graph containing $s$, and in doing so defines a spanning tree rooted at $s$.
- The path in the spanning tree from $s$ to $v$, corresponds to a shortest path in $G$.

Example of a spanning tree rooted at vertex 1.



# Breadth-First Search – Properties

Properties:

BFS/P1 For each $j \geq 1$, layer $L_j$ produced by BFS consists of all nodes at distance exactly $j$ from $s$.

BFS/P2 There is a path from $s$ to $t$ if and only if $t$ appears in some layer.

BFS/P3 Let $T$ be a breadth-first spanning tree, let $u, v$ be vertices in $T$ belonging to layers $L_i$ and $L_j$ respectively, and let $(u, v)$ be an edge of $G$. Then $i$ and $j$ differ by at most 1.

# Implementing Breadth-First Search
# Data Structures

- The adjacency list data structure is ideal for implementing a BFS algorithm.
- The algorithm examines the edges incident on a given vertex $u$ one by one using its adjacency list **Adj**[u].
- Array `Discovered` of length $n$ stores whether or not vertex $u$ has been previously discovered by the search.
- To maintain the vertices in a layer $L_i$, we have a list `L[i]`, for each $i = 0, 1, 2, \ldots$ and $i < n - 1$.

# Implementing Breadth-First Search/2

```
BFS(G,s)
    Discovered[s]=true;
    Discovered[u]=false, for all other u ∈ V;
    L[0]=s; layer counter i=0; spanning tree T=s;
    While L[i] ≠ ∅
      Initialize an empty list L[i+1]
      For each node u ∈ L[i]
        For each edge (u,v) incident to u;
          If Discovered[v]=false then
            Discovered[v]=true;
            Add edge (u,v) to tree T;
            Add v to the list L[i+1]
          EndIf
        EndFor
      EndFor
      i=i+1;
    Endwhile
```

# BFS Complexity

- The inner `For-Loop` takes $O(\mathbf{deg}(u))$ time for each vertex $u$;
- Thus, in total we need $O(\Sigma_{u \in V} \mathbf{deg}(u))$;
- From graph properties, $\Sigma_{u \in V} \mathbf{deg}(u) = 2 \cdot m$;
- Thus, $O(\Sigma_{u \in V} \mathbf{deg}(u)) = O(m)$;
- We need $O(n)$ additional time to set up lists and manage the array `Discovered`;
- Finally, the BFS runs in $O(m + n)$.

# Depth-First Search (DFS)

- BFS visits vertices at increasing distances: starts with distance 1 from $s$, then those at distance 2, and so on.
- Depth-First Search (DFS): follows some path as deeply as possible into the graph before it is forced to backtrack.
- BFS and DFS both build the connected component containing $s$ with a similar complexity.

# DFS — Recursive version

```
DFS(G,u)
Explored[u]=true;
If u ≠ s add edge (parent[u],u) to T;
for each edge (u,v) incident to u do
    if Explored[v]=false then
        parent[v] = u;
        DFS(G,v)
```

To apply this to *s-t connectivity*, we:

- Declare all vertices initially to be not explored;
- Initialize $\mathcal{T}$ to be a tree with root $s$;
- Invoke `DFS(G,s)`.

# Depth-First Search Tree

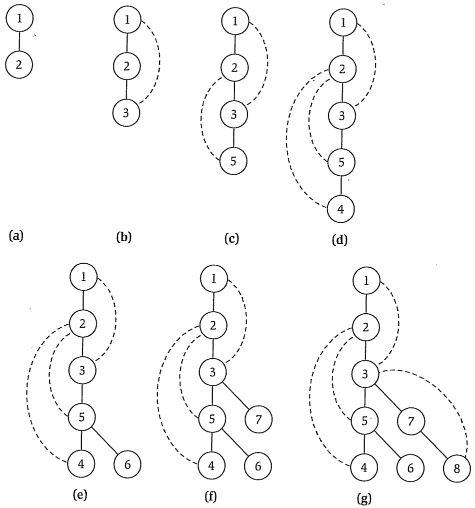



**Figure 3.5** The construction of a depth-first search tree $T$ for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of $T$; the dotted edges are edges of $G$ that do not belong to $T$.

# Depth-First Search Tree/2

- The spanning tree, also called depth-first search tree, generated by the DFS has a very different structure.
- The starting vertex $s$ is the root of $T$;
- A vertex $v$ is a child of $u$ in $T$ if `DFS(G,v)` is called directly during the call `DFS(G,u)`.

# DFS — Properties

DFS/P1 For a given recursive call `DFS(G,u)`, all nodes that are marked "Explored" between the invocation and the end of this recursive call are descendants of $u$ in $T$.

DFS/P2 Let $T$ be a depth-first search tree, let $u$ and $v$ be two nodes in $T$, and let $(u, v)$ be an edge of $G$ that is not an edge of $T$. Then one of $u$ or $v$ is an ancestor of the other.

# Depth-First Search – Iterative Algorithm

- Maintain the vertices to be processed in a stack: The recursive calls of DFS can be viewed as pushing vertices into a stack for later processing.

```
DFS(G,s)
Initialize S to be a stack with one element s;
Initialize T to be a tree with root s;
Initialize Explored[u] = false, for all v ∈ V;
while S ≠ ∅ do
    Pop a node u from S;
    if Explored[u] = false then
        Explored[u] = true;
        If u ≠ s add edge (parent[u],u) to T;
        for each edge (u, v) incident to u do
            Push v to the stack S;
            parent[v] = u
```

# DFS Complexity

- The main step in the algorithm is to push and pop vertices to and from the stack $S$;
- How many elements ever get pushed (and thus popped) to $S$?
- Vertex $v$ will be pushed to the stack $S$ every time one of its $\mathbf{deg}(v)$ adjacent vertices is explored.
- Thus, in total we need $O(\Sigma_{u \in V} \mathbf{deg}(u)) = O(m)$;
- We need $O(n)$ additional time to manage the array `Explored`;
- Finally, the DFS runs in $O(m + n)$.

# The Set of All Connected Components

Property: For any two nodes $s$ and $t$ in a graph, their connected components are either identical or disjoint.

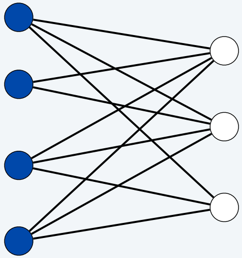
To compute all connected components of a graph $G$:

1. Start with an arbitrary node $s$, and, using BFS or DFS, generate its connected component;
2. Find a node $v$ (if any) that was not visited by the previous search, and generate its connected component–which will be disjoint from the previous components.
3. Continue till all vertices have been visited.

# Bipartite Graphs – 2-Colorability

Bipartite Graph: An undirected graph $G = (V, E)$ is Bipartite (or, 2-Colorable) if the vertices can be colored blue or white such that every edge has one white and one blue end.

- Applications.
  Matching: residents = blue, hospitals = white;
  Scheduling: machines = blue, jobs = white.

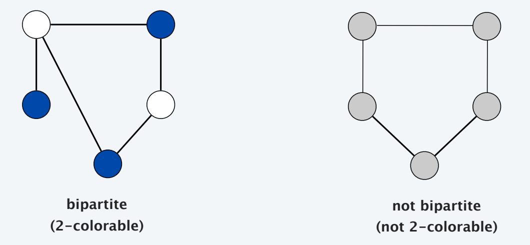


a bipartite graph

# Bipartite Graphs – 2-Colorability/2

What can be an obstacle for a graph not to be bipartite?

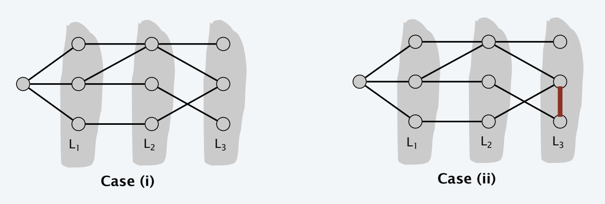
- For example, a triangle is not bipartite.
- Property. If a graph G is bipartite, it cannot contain an odd-length cycle.



# Bipartite Graphs – Property

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

1. No edge of $G$ connects two vertices of the same layer, and $G$ is bipartite.
2. An edge of $G$ connects two vertices of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

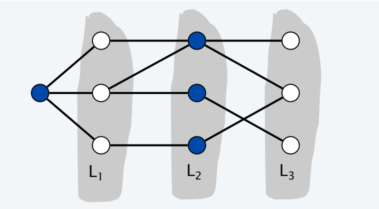


# Bipartite Graphs – Property/2

1. No edge of $G$ connects two vertices of the same layer, and $G$ is bipartite.

Proof of (1).

- Suppose no edge connects two vertices in same layer.
- By [BFS/P3] property, each edge of the Graph connects two vertices in adjacent levels.
- Bipartition (2-Coloring): blue = vertices on even levels, white = vertices on odd levels.

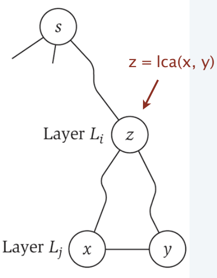


# Bipartite Graphs – Property/3

1. No edge of $G$ connects two vertices of the same layer, and $G$ is bipartite.
2. An edge of $G$ connects two vertices of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

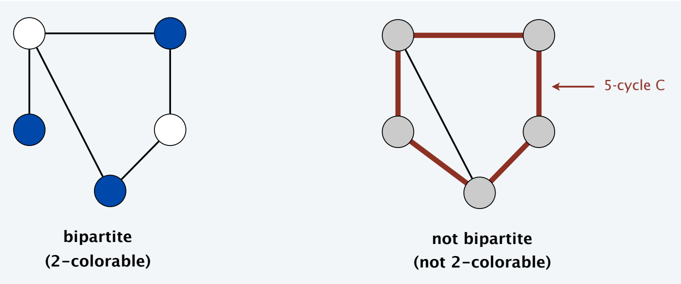
Pf. (ii)

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



# The Only Obstruction to Bipartiteness

Corollary. A graph $G$ is bipartite iff it contains no odd-length cycles.




Complexity of Bipartiteness: $O(m+n)$.

What about the Algorithm?

# Thank You!