Algorithms for Data Science
CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Graphs, Breadth-First Search (BFS)

Outline

1. Graphs

2. Representing graphs

3. Breadth-first search (BFS)

4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Graphs

Definition 1.

A **directed** graph consists of a finite set V of vertices (or nodes) and a set E of directed edges. A directed edge is an ordered pair of vertices (u, v).

- In mathematical terms, a directed graph $G = (V, E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V.
- An **undirected** graph is the special case of a directed graph where $(u, v) \in E$ if and only if $(v, u) \in E$. In this case, an edge may be indicated as the unordered pair $\{u, v\}$.
- **Notational conventions:** $|V| = n$, $|E| = m$

Node degrees

- **Undirected** graphs

$$deg(v) \triangleq \text{ number of edges incident to } v$$

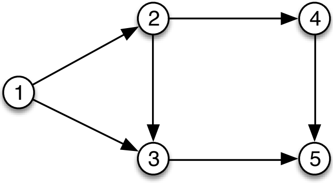
- **Directed** graphs

$$\begin{aligned} indeg(v) &\triangleq \text{ number of edges entering } v \\ outdeg(v) &\triangleq \text{ number of edges leaving } v \end{aligned}$$

Example graphs

Circles denote vertices (nodes).
Lines denote edges connecting vertices.
Arrows on lines indicate the direction along which the edge may be traversed.

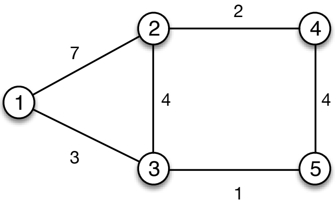
A directed, unweighted graph G
(default edge weight w(e) = 1)



indeg(1) = 0
indeg(3) = 2

outdeg(1) = 2
outdeg(3) = 1

An undirected, weighted graph G’



deg(1) = 2
deg(3) = 3

Examples of graphs (networks)

- **Transportation** networks: e.g., nodes are cities, edges (potentially weighted) are highways connecting the cities
    - *Can we reach a city j from a city i?*
    - *If* **yes**, *what is the shortest (or cheapest) path?*
- **Information** networks: e.g., the World Wide Web can be modeled as a directed graph
- **Wireless** networks: nodes are devices sitting at locations in physical space and there is an edge from u to v if v is close enough to u to hear from it.
- **Social** networks: e.g., nodes are people, edges represent friendship
- **Dependency** networks: e.g., given a list of functions in a large program, find an order to test the functions.

Useful definitions

- A **path** is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that consecutive vertices are adjacent, that is, there exists an edge $(x_i, x_i + 1) \in E$ for all $1 \leq i \leq n - 1$.

  Example: $(1, 2, 3, 2, 4)$ in G' is a path.

- A path is **simple** when all vertices are distinct.

  Example: $(1, 2, 4)$ in G' is a simple path.

- A **simple cycle** is a simple path that ends where it starts $(x_n = x_1)$.

  Example: $(1, 2, 3, 1)$ in G' is a simple cycle.

Useful definitions (cont'd)

The **distance** from u to v, denoted by $dist(u, v)$, is the *length* of the *shortest* path from u to v.

- **Unweighted** graphs:

  $$\textit{length} \text{ of path } P \triangleq \# \text{ edges on } P$$

- **Weighted** graphs: *coming up in a few lectures.*

- *Shortest* path from u to v: a path of minimum length among all paths from u to v.

Example: in G, $dist(1, 4) = 2$.

Useful definitions (cont'd)

- An undirected graph is **connected** when there is a path between every pair of vertices.

  Example: G' is connected.

- The **connected component** of a node u is the set of all nodes in the graph reachable by a path from u.

  Example: the connected component of 1 in G' is $\{1, 2, 3, 4, 5\}$.

- A directed graph is **strongly connected** if for every pair of vertices u, v, there is a path from u to v and from v to u.

- The **strongly connected component** of a node u in a directed graph is the set of nodes v in the graph such that there is a path from u to v and from v to u.

  Example: the strongly connected component of 1 in G is $\{1\}$.

Trees and tree properties

Definition 2.

A tree is a **connected acyclic** graph (undirected graphs). Or;
A **rooted** graph such that there is a unique path from the root to any other vertex (all graphs).

A tree is the most widely used special type of graph: it is the minimal connected graph.

Lemma 3.

Let G be an undirected graph. Any two of the following properties imply the third property, and that G is a tree.

1. *G is connected;*
2. *G is acyclic;*
3. $|E| = |V| - 1$.

Matchings and bipartite graphs

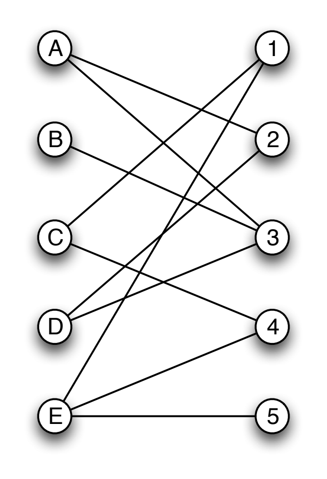
Bipartite graphs: vertices can be split into two subsets such that there are no edges between vertices in the same subset.

- Applications: social networks, coding theory
- **Notation:** $G = (X \cup Y, E)$, where $X \cup Y$ is the set of vertices in G and every edge in E has one endpoint in X and one endpoint in Y.

Example: suppose there are 5 people and 5 jobs and certain people qualify for certain jobs.

Matchings in bipartite graphs

Matching: a subset of the edges where every node appears at most once.



Goal: find a **one-to-one matching** (also called, a perfect matching) of people to jobs, if one exists.

Degree theorem

Theorem 4.

In any graph, the sum of the degrees of all vertices is equal to twice the number of the edges.

Proof.

Every edge is incident to two vertices, thus contributes twice to the total sum of the degrees. (Summing the degrees of all vertices simply counts all instances of some edge being incident to some vertex.) □

Running time of graph algorithms

Input: graph $G = (V, E), |V| = n, |E| = m$

- Linear graph algorithms run in $O(n + m)$ time
  - *Lower bound on m (assume connected graphs)?*
  - *Upper bound on m (assume simple graphs)?*

- More general running times: the best performance is determined by the relationship between n and m
  - For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Representing graphs: adjacency matrix

We want to represent a graph $G = (V, E)$, $|V| = n$, $|E| = m$.

Adjacency matrix for G: an $n \times n$ matrix A such that

$$A[i,j] = \begin{cases} 1, & \text{if edge } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Space required for adjacency matrix A: $\Theta(n^2)$.

Remark 1.

Space requirements can be improved if the graph is

- *undirected: A is symmetric ⇒ only store its upper triangle*
- *unweighted: only need 1 bit per entry*

Pros/cons of adjacency matrix representation

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ by its adjacency matrix has the following pros/cons.

Advantages:

1. check whether edge $e \in E$ in constant time
2. easy to adapt if the graph is weighted
3. suitable for dense graphs where $m = \Theta(n^2)$

Drawbacks:

1. requires $\Omega(n^2)$ space even if G is sparse $(m = o(n^2))$.
2. does not allow for linear time algorithms in sparse graphs (at least when all matrix entries must be examined).

Representing graphs: adjacency list

An alternative representation for graph $G = (V, E)$, $|V| = n$, $|E| = m$ is as follows.

Adjacency list: recall that vertex j is adjacent to vertex i if $(i, j) \in E$; then the adjacency list for vertex i is simply the list of vertices adjacent to vertex i.

The adjacency list representation of a graph consists of an array A with n entries such that $A[i]$ points to the adjacency list of vertex i.

Space requirements for adjacency list

Need

- an array of n pointers: $O(n)$ space; plus
- the sum of the lengths of all adjacency lists:
  - directed G: maintain the list of vertices with incoming edges from v and the list of vertices with outgoing edges to v.
    - length of adjacency lists of $v = outdeg(v) + indeg(v)$
    - length of all adjacency lists $= \sum_v outdeg(v) + indeg(v) = 2m$
  - undirected G: maintain the list of vertices adjacent to v
    - length of adjacency list of $v = deg(v)$
    - length of all adjacency lists $= \sum_v deg(v) = 2m$.

$\Rightarrow$ Total space: $O(n + m)$

Pros/cons of representing G by its adjacency list

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ using adjacency lists has the following pros/cons.

Advantages:

- allocates no unnecessary space: $O(n + m)$ space to represent a graph on n vertices and m edges
- suitable for linear or near-linear time algorithms

Drawbacks:

- searching for an edge can take $O(n)$ time

Adjacency list vs Adjacency matrix

We prefer **adjacency matrix** when

- we need determine quickly whether an edge is in the graph
- the graph is dense
- the graph is small (it is a simpler representation).

We use an **adjacency list** otherwise.

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Searching a graph

Given a transportation network and a city s, we want to find all cities reachable from s.

This problem is known as s-t connectivity.

Input: a graph $G = (V, E)$, a vertex $s \in V$

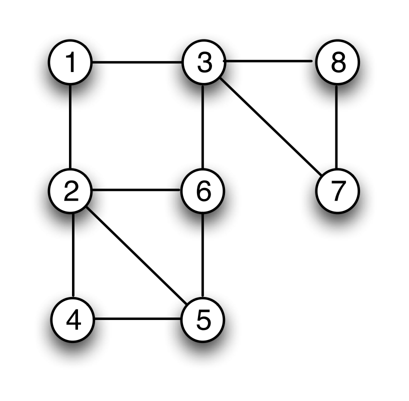
Output: all vertices $t \in V$ such that there is a path from s to t

An algorithm for s-t connectivity: breadth-first search

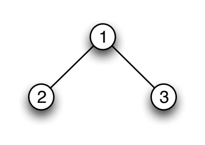
Breadth-first search (BFS): explore G starting from s **outward in all possible directions**, adding reachable nodes one **layer** at a time.

- First add all nodes that are joined by an edge to s: these nodes form the first layer.
  If G is unweighted, these are the nodes at distance 1 from s.
- Then add all nodes that are joined by an edge to a node in the first layer: these nodes form the second layer.
  If G is unweighted, these are the nodes at distance 2 from s.
- And so on and so forth.

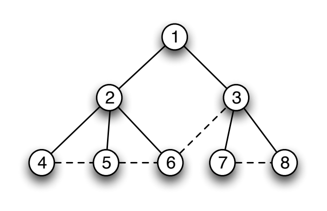
Example graph G_1



The BFS layers for the example graph G_1



1st layer



2nd layer

Solid edges appear in G_1 and in the output of BFS.

Dotted edges appear in G_1 but do not in the output of BFS.

Ties are broken by selecting the node with the smallest index.

Properties of the layers of the output of BFS

Formally,

- **Layer** L_0 contains s.
- **Layer** L_1 contains all nodes v such that $(s, v) \in E$.
- For $i \geq 1$, **layer** L_i contains all nodes that
  1. have an edge from a node in layer L_{i-1}; and
  2. do **not** belong to a previous layer.

Fact 5.

L_i is the set of nodes that are at distance i from s.

Equivalently, the length of the shortest s-v path for all $v \in L_i$ equals i.

Proof of Fact 5

By induction.

- **Basis:** true for layer L_0.
- **Hypothesis:** suppose L_i is the set of nodes at distance i from s, for some $i \geq 0$.
- **Step:** The only vertices added to L_{i+1} are those that
  1. have an edge from a node in L_i; and
  2. do not have an edge from a node in **any previous** layer L_k, for $k < i$

  Then L_{i+1} contains the nodes that are at distance $1 + i$ from s.

Output of BFS

- *When is a node v added to the graph produced by BFS?*
- *Why would a node fail to appear in the BFS output?*
- *Which problems are solved by BFS?*
- *Why is the graph produced by BFS a tree?*
- *Consider an edge $(u, v) \in E$ that does* **not** *appear in the BFS tree. u, v must appear at some layers of the BFS tree, say L_i, L_j respectively. How far apart can these layers be?*

Output of BFS

- *When is a node v reachable from s added to the BFS graph?*

  A node v is added to the tree when it is *discovered*, that is, when some node u is being *explored* and an edge (u, v) is found for the first time. Then u becomes the parent of v since u is responsible for *discovering* v.

- *Why would a node fail to appear in the BFS graph?*

  Because there is no path from s to that node.

BFS in G started at vertex s answers

1. s-t connectivity;
2. shortest s-v paths in *unweighted* graphs.

Properties of the BFS tree (cont'd)

- *Why is the graph produced by BFS a tree?*

  Because it is connected and acyclic.

  - The order in which the vertices are visited matters for the final tree but **not** for the distances computed.

- *What are graph edges that do not appear in the BFS tree?*

  They are either

  - edges between nodes in the same layer; or
  - edges between nodes in adjacent layers.

  This is a property of **all** BFS trees.

Non-tree edges in BFS trees

Claim 1.

Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge in G. Then i and j differ by at most 1.

Proof of Claim 1

Proof.

Without loss of generality, assume that x was *discovered* **first** during execution of BFS. Therefore, $i \leq j$ and x is *explored* before y. When x is *explored*, there are two possibilities.

1. y is *discovered* then, hence y is added at layer L_{i+1} as a child of x. Or;
2. y was *discovered* **before** x is *explored*. Thus y appears in the tree at some layer L_j with $j \leq i$. Since $j \geq i$, y must appear at layer L_i.

□

Implementing BFS

We need to store the nodes *discovered* at layer L_i in order to *explore* them later (once we have finished exploring layer L_i).

To this end we use a queue.

- FIFO data structure: add to the end of the queue, extract from the head of the queue.
- Implemented as a double-linked list: maintain explicit pointers to the head and tail elements. Then **enqueue** and **dequeue** operations take constant time.

Pseudocode for BFS —running time?

```
BFS($G = (V, E), s \in V$)
    array $discovered[V]$ initialized to 0
    array $dist[V]$ initialized to $\infty$
    array $parent[V]$ initialized to $NIL$
    queue $q$
    $discovered[s] = 1$
    $dist[s] = 0$s
    $parent[s] = NIL$
    enqueue($q, s$)
    while $size(q) > 0$ do
        $u =$dequeue($q$)
        for $(u, v) \in E$ do
            if $discovered[v] == 0$ then
                $discovered[v] = 1$
                $dist[v] = dist[u] + 1$
                $parent[v] = u$
                enqueue($q, v$)
            end if
        end for
    end while
```

Today

1. Graphs

2. Representing graphs

3. Breadth-first search (BFS)

4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Connected components in undirected graphs

- BFS(s) naturally produces the connected component $R(s)$ of vertex s, that is, the set of nodes reachable from s.
  - Exploring the vertices in a different *order* can yield different algorithms for finding connected components (*coming up in the next lecture*).

- *How can we produce all the connected components of G?*
  - Consider two distinct vertices s and t in G: *how do their connected components compare?*

Connected components of different vertices

Fact 6.

For any two vertices u and v their connected components are either the same or disjoint.

Proof.

Consider any two nodes s, t such that there is a path between them: their connected components are the same (*why?*).

Now consider any two nodes s, t such that there is no path between them: their connected components are disjoint. If not, there is a node v that belongs to both components, hence a path between s and v and a path between t and v. Then there is a path between s and t, contradiction. □

Finding all the connected components in a graph

`AllConnectedComponents`$(G = (V, E))$

1. Start with an arbitrary node s; run `BFS`(G, s) and output the resulting BFS tree as one connected component.

2. Continue with any node u that has not been visited by `BFS`(G, s); run BFS from u and output the resulting BFS tree as one connected component.

3. Repeat until all nodes in V have been visited.

Testing bipartiteness & graph 2-colorability

Testing bipartiteness

- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if G is bipartite, **no** otherwise

Equivalent problem (*why?*)

- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if and only if we can color all the vertices in G using at most 2 colors –say red and white– so that no edge has two endpoints with the same color.

Why wouldn't we be able to 2-color a graph?

Fact: If a graph contains an odd-length cycle, then it is not 2-colorable.

So a **necessary** condition for a graph to be 2-colorable is that it does not contain odd-length cycles.

Is this condition also **sufficient***, that is, if a graph does not contain odd-length cycles, then is it 2-colorable?*

In other words, are odd cycles the only obstacle to bipartiteness?

Algorithm for 2-colorability

BFS provides a natural way to 2-color a graph $G = (V, E)$:

- Start BFS from any vertex; color it red.
- Color white all nodes in the first layer L_1 of the BFS tree. If there is an edge between two nodes in L_1, output **no** and stop.
- Otherwise, continue from layer L_1, coloring red the vertices in even layers and white in odd layers.
- If BFS terminates and all nodes in V have been explored (hence 2-colored), output **yes**.

Analyzing the algorithm

Upon termination of the algorithm

- either we successfully 2-colored all vertices and output **yes**, that is, declared the graph bipartite;
- or we stopped at some level because there was an edge between two vertices of that level and output **no**; in this case, we declared the graph non-bipartite.

This algorithm is **efficient**. *Is it a* **correct** *algorithm for 2-colorability?*

Showing correctness

To prove correctness, we must show the following statement: if our algorithm outputs

1. **yes**, the 2-coloring it provides is a valid 2-coloring of G;
2. **no**, G is indeed not 2-colorable by **any** algorithm (e.g., because it contains an odd-length cycle).

The next claim proves that this is indeed the case by examining when the algorithm succeeds to 2-color G. Recall that the only reason why the algorithm fails to 2-color G is because it found an edge between two nodes of the same layer.

Correctness of algorithm for 2-colorability

Claim 2.

Let G be a connected graph, and let L_1, L_2, ... be the layers produced by BFS starting at node s. Then exactly one of the following two is true.

1. *There is no edge of G joining two nodes of the same layer.* *Then G is bipartite and has no odd length cycles.*

2. *There is an edge of G joining two nodes of the same layer.* *Then G contains an odd length cycle, hence is not bipartite.*

Corollary 7.

A graph is bipartite if and only if it contains no odd length cycle.

Proof of Claim 2, part 1

1. **Assume** that no edge in G joins two nodes of the same layer of the BFS tree.

By Claim 1, all edges in G not belonging to the BFS tree are

- either edges between nodes in the same layer;
- or edges between nodes in adjacent layers.

Our assumption implies that all edges of G not appearing in the BFS tree are between nodes in adjacent layers.

Since our coloring procedure gives such nodes different colors, the whole graph can be 2-colored, hence it is bipartite.

Proof of Claim 2, part 2

2. **Assume** that there is an edge $(u, v) \in E$ between two nodes u and v in the same layer.

Obviously G is not 2-colorable by our algorithm: both endpoints of edge (u, v) are assigned the same color.

Our algorithm returns **no**, hence declares G non-bipartite.

Can we show existence of an odd-length cycle and prove that G indeed is not 2-colorable by **any** *algorithm?*

Proof of correctness, part 2

- Let u, v appear at layer L_j and edge $(u, v) \in E$.
- Let z be the lowest common ancestor of u and v in the BFS tree (z might be s). Suppose z appears at layer L_i with $i < j$.
- Consider the following path in G: from z to u follow edges of the BFS tree, then edge (u, v) and back to z following edges of the BFS tree. This is a cycle starting and ending at z, consisting of $(j - i) + 1 + (j - i) = 2(j - i) + 1$ edges, hence of odd length.