Graphs

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

November 4, 2024

Graphs and Their Representations

Graphs

- A **graph** is a data structure which encodes pairwise relationships among a set of objects.
- A graph $G = (V, E)$ is a collection of vertices V and a collection of edges $E \subseteq V \times V$. Each edge represents a relationship between the corresponding vertices in V.
- In some settings, the relationship that an edge represents is symmetric, and thus an edge is just a subset of two vertices $\{u, v\}$ for some $u, v \in V$ (we call u and v *neighbors*). Othertimes the relationship is asymmetric, and thus an edge is an ordered pair (u, v).

Directed and Undirected Graph

- If the edges represent asymmetric relationships, then we call the graph a *directed graph* (or *digraph*). If a graph is not specified to be directed, then generally we view the edges as representing symmetric relationships. Sometimes these graphs are called *undirected graphs*.
- In either case, we have $|E| = O(|V|^2)$.
- In an undirected graph, the *degree* of v is the number of neighbors of v. For digraphs, the *out degree* is the number of "outgoing" edges there are from v and the *in degree* is defined similarly.

Applications of Graphs

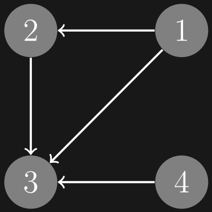
- There are many applications in which a graph may be a useful data structure:
  - Transportation networks
    - Vertices represent airports and edges represent the existence of a flight between the corresponding airports.
  - Communication networks
    - Vertices represent computers on a network and edges exist between computers with a direct physical link connecting them.
  - Information networks
    - Vertices represent web pages and (directed) edges represent a link from one web page to another.

Representing Graphs with Adjacency Matrix

- One way of representing a graph is an **adjacency matrix**. Let $|V| = n$. The adjacency matrix A of a graph G is the $n \times n$ matrix such that:

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

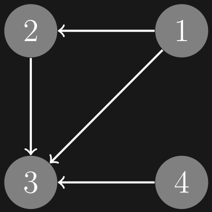
- An example of the adjacency matrix:



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Representing Graphs with Adjacency List

- Another way of representing a graph is an **adjacency list** representation. For each $v \in V$, its adjacency list $Adj[v]$ is the list of vertices adjacent to v.



$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$

- In an undirected graph, $|Adj[v]| = degree(v)$. For digraphs, $|Adj[v]| = out - degree(v)$.

Basic Properties of Graphs

- Handshaking Lemma:
  - Undirected graphs: $\sum_{v \in V} \mathit{degree}(v) = 2|E|$
  - Digraphs: $\sum_{v \in V} \mathit{in\text{-}degree}(v) + \sum_{v \in V} \mathit{out\text{-}degree}(v) = 2|E|$
- This implies that adjacency lists use $\Theta(|V| + |E|)$ storage. Contrast this with adjacency matrices use $O(|V|^2)$ storage. Adjacency lists use less storage when the graphs are "sparse" (sub-quadradic number of edges).
- We will assume we are using the adjacency list representation unless stated otherwise.

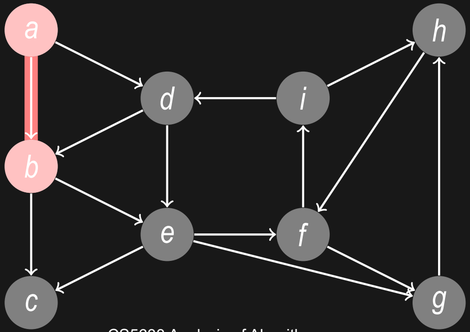
Graph Traversal Algorithms

Graph Traversal

- A common task one might want to do with a graph is to traverse each of the nodes in the graph.
- Two popular graph traversal algorithms:
  - Breadth-first search (BFS): Start from an arbitrary vertex v and visit the remaining vertices in “layers”. We first visit all of v’s neighbors (the first layer), and then we visit all of the neighbors of the first layer (which we have not already visisted), etc.
  - Depth-first search (DFS): Start from an arbitrary vertes v, and then visit one neighbor of v. Then visit a new neighbor from this vertex. Keep visiting an unvisited neighbor until we get “stuck”, then backtrack and try again.
- When discussing these algorithms, we assume $|V| = n$ and $|E| = m$ (common notation when discussing graphs).

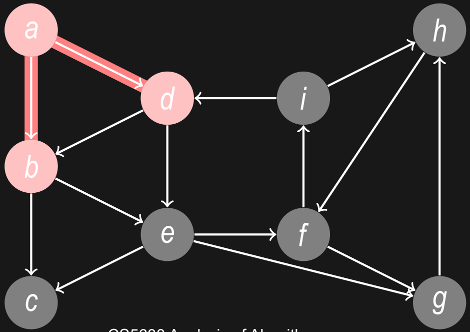
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



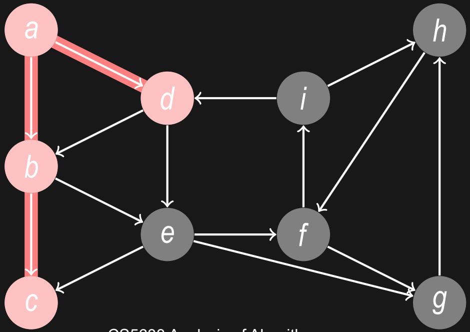
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



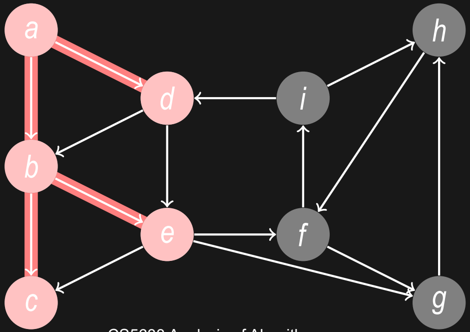
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



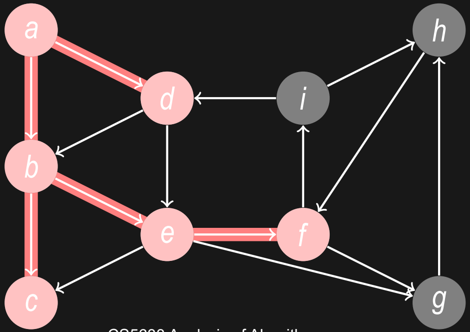
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



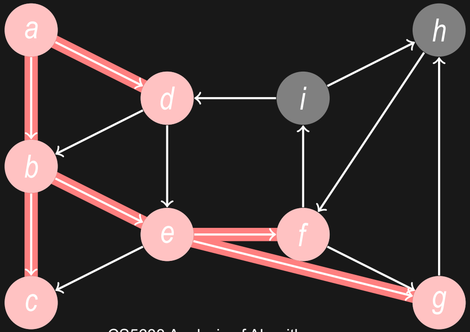
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



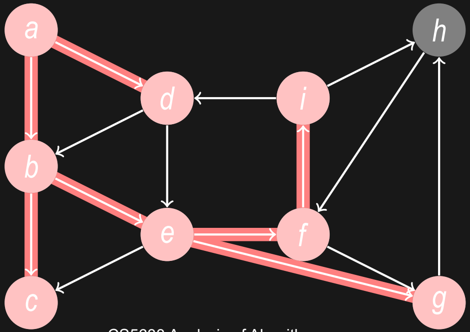
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



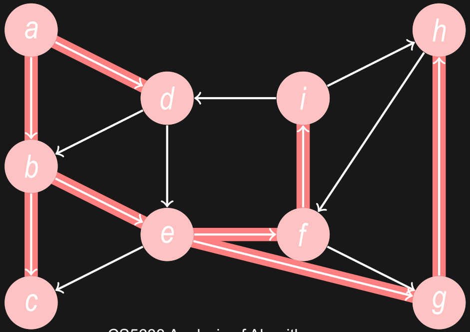
Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



Breadth-first Search (BFS)

- Breadth-first search: maintain a queue Q of nodes that we need to visit. We will compute a directed BFS tree T which remembers the order in which we visited the nodes in the graph.
- An example of BFS search. Red edges indicates the traversal path. The queue for this traversal is $Q = \{a, b, d, c, e, f, g, i, h\}$.



Implementation of Breadth-first Search with a Queue

Algorithm 1: BFS implementation with a queue.

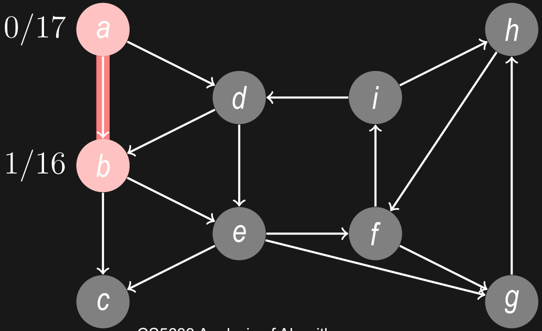
```
Function BFS(graph G)
    while There are unvisited vertices do
        Q = empty queue;
        choose an unvisited vertex a;
        Q.append(a);
        while Q is not empty do
            v = Q.remove_first();
            mark v as visited;
            for each neighbor w of v do
                if w is not visited then
                    Q.append(w);
```

Run-time of Breadth-first Search

- Each vertex is marked as visited at most once and enqueued/dequeued at most once ($O(|V|)$ time).
- Each edge is "checked" to see if the neighbor has been already visited visited twice ($O(|E|)$ time).
- Therefore the running time is $O(n + m) = O(|V| + |E|)$.

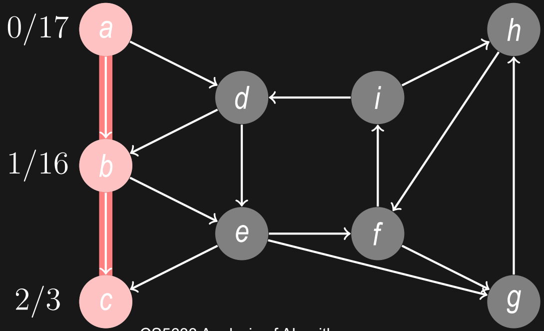
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a "discovery time" and "finishing time" as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\textit{discovery\_time}/\textit{finish\_time}$.



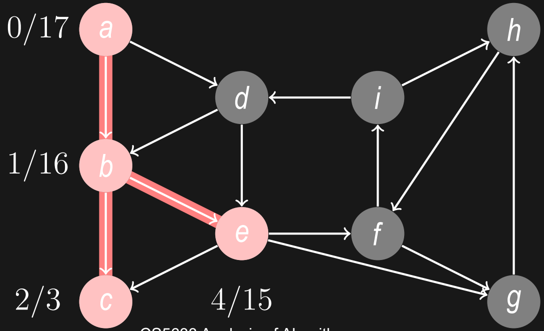
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a “discovery time” and “finishing time” as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\textit{discovery\_time}/\textit{finish\_time}$.



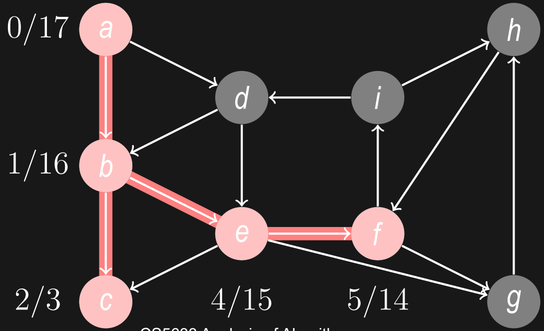
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a “discovery time” and “finishing time” as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $discovery\_time/finish\_time$.



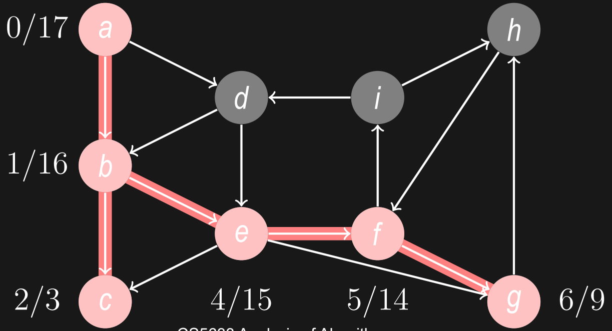
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a "discovery time" and "finishing time" as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\textit{discovery\_time}/\textit{finish\_time}$.



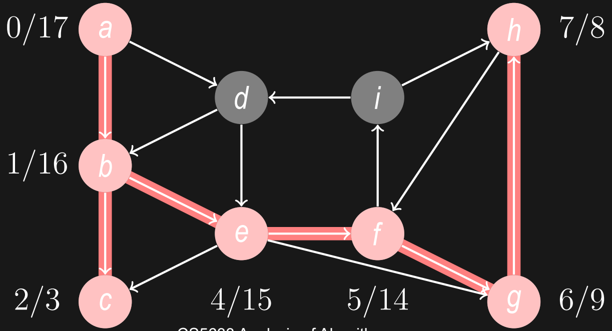
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a “discovery time” and “finishing time” as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\mathit{discovery\_time}/\mathit{finish\_time}$.



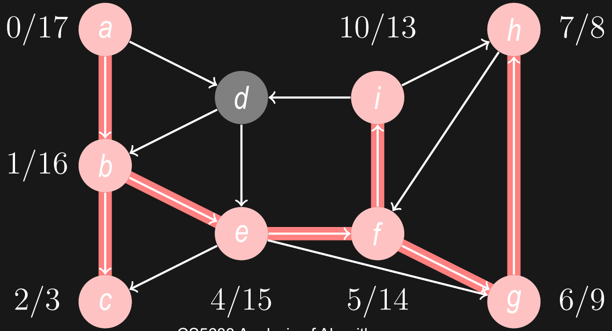
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a “discovery time” and “finishing time” as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\textit{discovery\_time}/\textit{finish\_time}$.



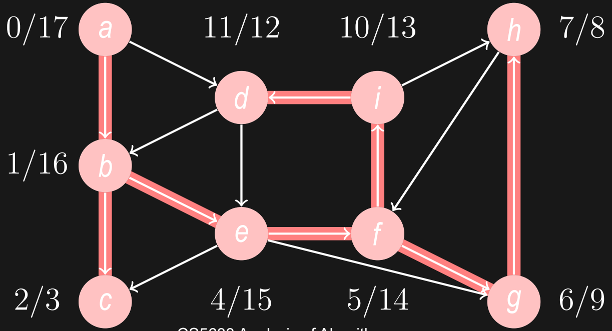
Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a "discovery time" and "finishing time" as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $discovery\_time/finish\_time$.



Depth-first Search (DFS)

- Depth-first search: look for a neighbor which we have not already visited. Once we find such a neighbor, immediately visit this neighbor. Maintain a "discovery time" and "finishing time" as well as a DFS tree T.
- An example of DFS search. Red edges indicates the traversal path. The number besides each node gives the $\mathit{discovery\_time}/\mathit{finish\_time}$.



Recursive Implementation of Depth-first Search

- DFS is usually implemented as a recursive function or with a stack.
- The recursive implementation.

Algorithm 2: Recursive DFS. Start with any vertex.

```
Function Recursive_DFS(vertex v, graph G)
    mark v as visited; // vertex v discovered;
    for each neighbor w of v do
        // recursively check each of v's neighbor;
        if w is not visited then
            Recursive_DFS(w, G);
    // vertex v finished;
```

Implementation of Depth-first Search with Stack

- The stack implementation.

Algorithm 3: DFS with a stack.

```
Function Stack_DFS(graph G)
    while There are unvisited vertices do
        s = empty stack;
        choose an unvisited vertex a;
        mark a as visited; // vertex a discovered;
        s.push(a);
        while s is not empty do
            v = s.top(); // note not popped;
            if all v's neighbor are visited then
                s.pop(); // vertex w finished;
            else
                choose a un-visited neighbor w;
                mark w as visited; // vertex w discovered;
                s.push(w);
```

Run-time of Depth-first Search

- Run-time of DFS is the same as BFS.
  - Each vertex is marked as visited at most once and enqueued/dequeued at most once ($O(|V|)$ time).
  - Each edge is “checked” to see if the neighbor has been already visited visited twice ($O(|E|)$ time).
  - Therefore the running time is $O(|V| + |E|)$.

Topological Sort

Cyclic and Acyclic Graphs

- Given a graph $G : (V, E)$,
  - A path is a sequence of vertices $\{v_1, v_2, \ldots, v_k\}$ such that $(v_i, v_{i+1}) \in E$, for all $i \in \{1, \ldots, k-1\}$.
  - A path is simple if all vertices in the path are unique.
  - A path is a cycle if $v_1 == v_k$.
  - A graph has no cycles is acyclic.
    - Undirected acyclic graphs are called forest.
    - Directed acyclic graphs are called DAG.

Topological Sort

- DAGs commonly occur in some applications. For example if the graph is representing precidence between certain objects (e.g. course prerequisites).
- Motivated by this, we may be interested in computing a linear ordering of the vertices so that all of the edges “go to the right” (courses left to right that can be taken in order without violating prerequisites).
- Such an ordering is called a **topological sort**.

Topological Sorting Algorithm

Algorithm 4: Kahn's Algorithm.

```
Function Topo_Sort(graph G)
    Q = empty queue;
    S = set of nodes with no incoming edges;
    while S is not empty do
        v = S.remove_first();
        Q.append(v); // add v to sorted list;
        for each neighbor w of v do
            remove edge (v, w);
            if w has no edges then
                S.append(w);
    return Q;
```