Data Structures (67109-2)- Recitation 8

21-25.5.2023

The Hebrew University of Jerusalem

Graphs are widely used data structures that have many applications and useful algorithms. If you ever encounter a problem in your life, try graphifying it, and chances are there is already a brilliant algorithm to solve it. Today, we will explore some of the most important and useful techniques for traversing graphs, including breadth-first search (BFS) and depth-first search (DFS). These algorithms can help us find paths, cycles, connected components, and other properties in graphs. Now that we have an overview, let's begin with some definitions. You can find more materials for this lecture at Dastflix.

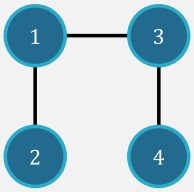
1 Graphs

1.1 Definitions, Examples, and Basics

Definition 1. undirected graph

A **undirected graph** G is defined as a pair of two sets: $G = (V, E)$, where V represents a set of vertices and E is a set of pairs of vertices, representing edges between the vertices.

Example: $G = (\{1, 2, 3, 4\},\ \{\{1, 2\}, \{1, 3\}, \{3, 4\}\})$ is the following graph:



Definition 2. degree of a vertex

The **degree of a vertex** of a undirected graph is the number of edges that are incident to the vertex. Explicitly defined as:

$$d(v) = |\{v \in V | (v, u) \in E\}|$$

The degree of a vertex can range from 0 to $n - 1$, where n is the number of vertices in the graph:

$$\forall v \in V \quad 0 \leq d(v) \leq n - 1$$

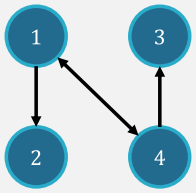
Additionally, there is a relationship between the sum of degrees of all vertices and the number of edges in a graph:

$$\sum_{v \in V} d(v) = 2|E|$$

Definition 3. directed graph

A **directed graph** G is defined as a pair of two sets: $G = (V, E)$, where V represents a set of vertices and $E \subseteq V \times V$ is a set of directed edges between vertices.

Example: $G = (\{1, 2, 3, 4\},\ \{(1, 2), (1, 4), (4, 1), (4, 3)\})$ is the following graph:



Here are some important points to mention:

- Short notation for edges: $e_{ij} = (v_i, v_j)$
- The number of edges ($|E|$) is at most the number of possible combinations of vertices $\binom{|V|}{2}$, which can be expressed as

$$|E| \leq \binom{|V|}{2} = \frac{|V|\,(|V| - 1)}{2} = O\left(|V|^2\right)$$

Definition 4. simple path

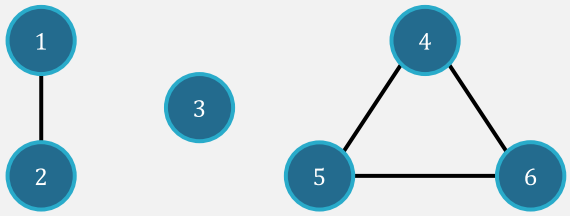
A **simple path** in graph G is a sequence of distinct vertices, denoted as $v_1, v_2, \ldots, v_k$, that are connected by edges in the given order. In other - $\forall i \in [k]\ \ \exists\,(v_i, v_{i+1}) \in E$.

The length of a path is determined by the number of edges it contains.

Definition 5. connectivity

1. Let $G = (V, E)$ be a undirected graph. A **connected component** of G is a subset $U \subseteq V$ of maximal size in which there exists a path between every two vertices.
2. A undirected graph G is considered **connected** if it has only one connected component.
3. Let $G = (V, E)$ be a directed graph. A **strongly connected component** of G is a subset $U \subseteq V$ of maximal size in which for any pair of vertices $u, v \in U$ there exist both directed path from u to v and a directed path form v to u.

Example: A graph with 3 connected components:



Definition 6. weighted graph

A **weighted graph**, denoted as $G = (V, E, w)$ is a graph where V represents the set of vertices, E represents the set of edges (either directed or undirected), and w is a weight function $w : E \to \mathbb{R}$. The weight function $w(e)$ is typically associated with the edge length, but it can have other interpretations as well.

Now that we can visually interpret graphs, we can envision various applications for them. For example, they can depict the relationships among friends on Facebook or illustrate how we navigate across various websites on the internet (The PageRank Algorithm effectively utilizes this perspective).

1.2 Graph Representation

How you store data structures in a computer can make or break your algorithm's efficiency. This is a hot topic of research - for example, people have been sweating hard to find clever ways to store matrices that can speed up matrix multiplication by a ε more. Graphs are no different - we need to store them smartly to keep the algorithms happy.

1.2.1 Adjacency-List Representation An adjacency list is a simple way to represent a graph as a list of vertices, where each vertex has a list of its buddies (Figure 1.1). That is, `Adj` $[u]$ contains all the vertices that u can reach in G. For both directed and undirected graphs, the adjacency-list representation has the nice feature that it only takes $\Theta(|V| + |E|)$ memory space. So, adjacency-list is a good choice for **sparse** graphs—those for which $|E| \ll |V|^2$.

1.2.2 Adjacency-Matrix Representation Adjacency Matrix is a 2D array of size $|V| \times |V|$ that tells you if there is an edge between two vertices or not:

$$a_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Obviously, it takes $\Theta\left(|V|^2\right)$ memory space. So, adjacency-matrix is a good choice for **dense** graphs—those for which $|E| \approx |V|^2$ or when you need to check quickly if two vertices are connected or not.

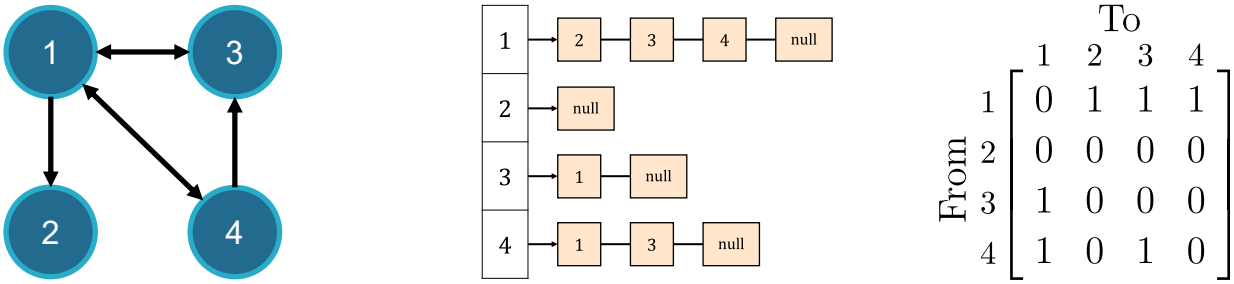


Fig. 1.1. Two ways to represent a directed graph. From left to right: A directed graph with 4 vertices and 6 edges, an adjacency-list representation of the graph, the adjacency-matrix representation of the graph.

2 Graphs Algorithms

2.1 Breadth First Search (BFS)

Once upon a time, two adventurous alumni of the HebrewU - Tom Sawyer and his old flame Becky - found themselves trapped in a dark cave with the notorious Injun Joe. But they were not afraid, for they knew that somewhere in the cave there was a hidden fabulous treasure waiting for them. Becky was a smart and savvy student who aced her data structures course. She quickly recalled the BFS algorithm that could help them explore the cave efficiently. Tom was a kind soul but a forgetful one, so Becky reminded him what they had to do: "Graph-Search algorithms are a fancy way of saying we need to leave some bread crumbs behind us as we go along. That way, we can check out every nook and cranny of the cave - and if we see some bread crumbs there, we know we've been there before and we can skip it." Tom and Becky wasted no time and followed Becky's plan. They were amazed to discover that they searched every corner and after $O(|E| + |V|)$ minutes they found the treasure and the way out. Now they are filthy rich and live under a false identity in Katamon.

As Becky mentioned, Breadth-First Search (BFS) is just one of the algorithms (DFS coming soon...) that can search through all the vertices of a graph. The characteristic of BFS is the order in which it visits the vertices (breadth-first order) and the creation of a breadth-first tree using the paths during the algorithm.

2.1.1 Shortest Path to Root in Unweighted Graph In the lecture, we saw how BFS found all the vertices, and how to track the breadth-first tree that the BFS path created. We introduce $\pi[v]$ and update it during the algorithm. $\pi[v]$ represents v's parent in the breadth-first tree. If v has no parent because it is the source vertex or is undiscovered, then $\pi[v]$ is None. The parent links trace the shortest path back to the root.

Algorithm 1 BFS

Input: A graph G and a starting vertex s.
Output: The parent links trace the shortest path back to s and the distances to s.

```
1  Function BFS(G,s):
2      queue ← Build-Queue({s})
3      for v ∈ V do
4          dist[u] ← ∞
5      dist[s] ← 0
6      π[s] ←None
7
8      while queue ≠ ∅ do
9          u = queue.pop(0)
10         for neighbor v of u and dist[v] = ∞ do
11             queue.push(v)
12             dist[v] ← dist[u] + 1
13             π[v] ← u
```

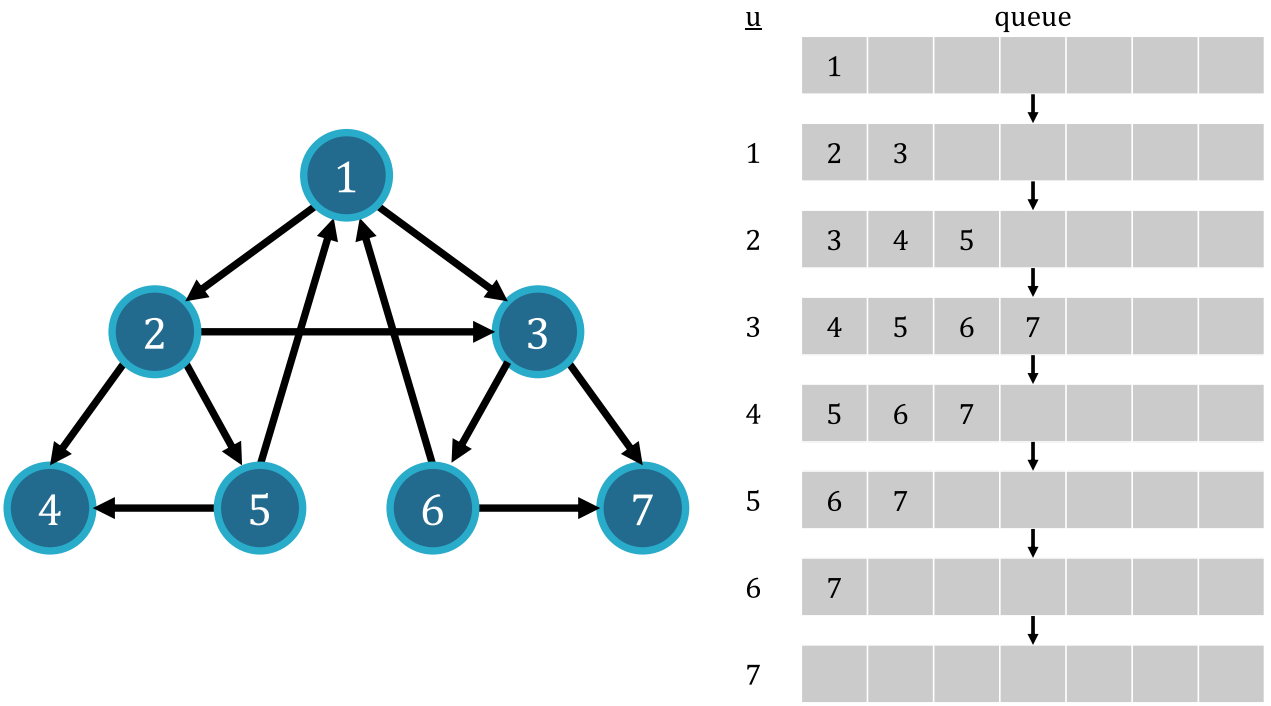


Fig. 2.1. An example of a BFS algorithm (Alg. 1) applied to a directed graph. The algorithm starts at the first floor and visits all vertices at that level before moving on to the next level. This process continues until all vertices have been visited.

2.1.2 BFS Time Complexity The BFS analysis is diffe'rent from what we have seen before. Here we will analyze the work of each loop separately, regardless of whether it is an inner or outer loop. To search for the neighbors of a vertex in O (number of neighbors) time, we need to use adjacency-list representation (sec. 1.2.1) for BFS (make sure you understand why).

- Lines 2-3: $|V|$ operations for initializing `dist` - $\Theta(|V|)$.
- line 8: The number of iterations that the `while` loop does is $\Theta(|V|)$. This is because every vertex can only enter the queue **ONCE** (since it is then marked as "visited"), and therefore it runs $|V|$ times.
- line 10: The number of iterations that the `for` loop does is $O(|E|)$. To understand this, we use the same technique as before - each vertex will only reach this for loop once. We search through all the vertex's neighbors - which are the edges out of this vertex. That's why we will iterate this for loop at most $O(|E|)$ times.

So the total runtime is $O(|E| + |V|)$

Note 1. BFS can also be applied to a forest (a graph with more than one connected component). To do that, we need to run BFS for each unvisited vertex until no unvisited vertex remains.

Note 2. BFS can work for any weighted graph with unit weights. However, if the weights are not uniform, we need different algorithms to find the shortest path. One example is Dijkstra's algorithm, which we will discuss later.

2.2 Depth First Search (DFS)

DFS is a way to explore all the vertices in a graph. Unlike BFS, which visits all the neighbors of a vertex before moving on, DFS dives deeper into the graph as soon as it finds a neighbor. This makes DFS walk in a different style than BFS - it goes as far as it can into the graph.

2.2.1 Non-Recursive DFS The non-recursive version of DFS is very similar to BFS (Alg. 1). The only difference is in line 9: `queue.pop()`. This simple change makes us go to the first neighbor we see, instead of the last one. That's it. Now we go deep inside the graph first.

2.2.2 Recursive DFS DFS can be more useful if we keep track of some extra information, such as the order of visiting the vertices. We can do this by assigning pre-visit and post-visit numbers to each vertex. These numbers tell us when a vertex enters and exits the recursion stack of DFS. Pre and post numbers are handy for many graph algorithms. For example, they can help us find out if a vertex belongs to the sub-tree of another vertex (as we will see later).

To do this, we need to use the Recursive DFS, which is shown below (Alg. 2).

Algorithm 2 DFS

Input: A graph G and a starting vertex s.
Output: The order of discovery (pre) and completion (post) for each vertex in G.

```
1 Function DFS(G,s):
2     s.visited ← True
3     time += 1
4     pre[s] ← time
5     for neighbor v of s and v.visited = False do
6         π[v] ← s
7         DFS(G,v)
8     time += 1
9     post[s] ← time
```

Let's demonstrate an example run of Recursive DFS (Figure 2.2):

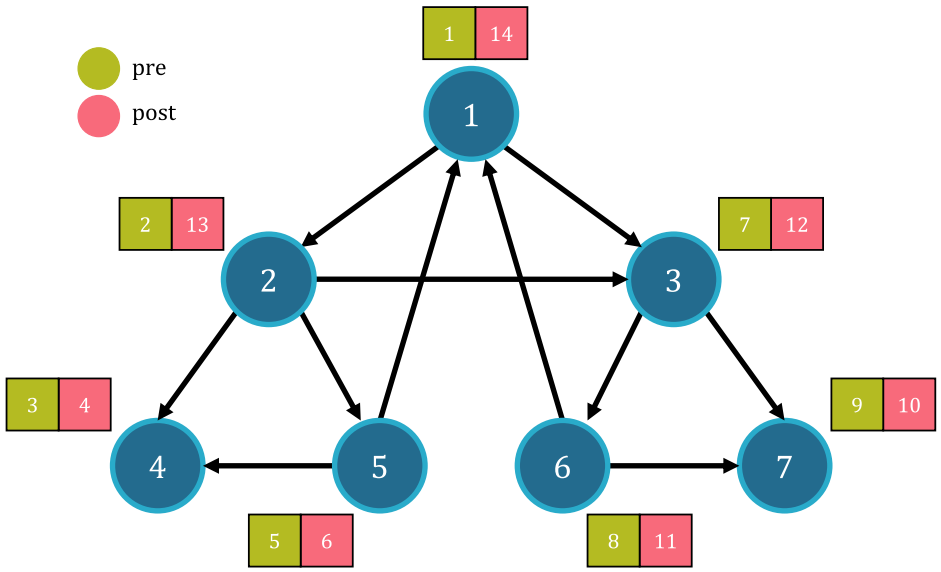


Fig. 2.2. An example run of DFS. We can see that DFS goes directly to the deepest node 4. At each sub-tree of the graph, DFS does the same - first visit the deepest vertices.

2.2.3 Properties of Depth First Search DFS provides us with important information. At first glance, it may not be clear why we should check-in and out each vertex. We will explain how it can be used to determine if a vertex is in the sub-tree of another vertex. To find out whether u lies in the sub-tree of v or not, we just compare the `pre` and `post` numbers of u and v. If $\texttt{pre}[v] < \texttt{pre}[u] < \texttt{post}[u] < \texttt{post}[v]$, then u lies in the sub-tree of v; otherwise, it does not. This distinction is the essence of the parenthesis theorem (Theorem 1). You can see an example below for more clarification (Figure 2.3).

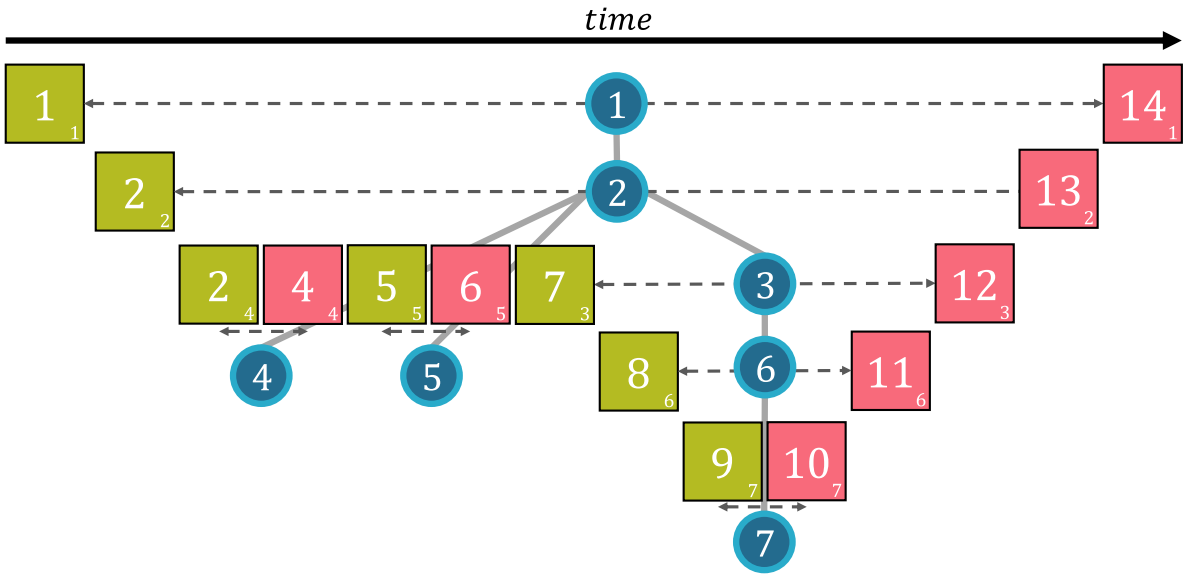


Fig. 2.3. Illustration of the Parenthesis Theorem using the example shown in Figure 2.2. It is evident that all intervals of the vertices are completely contained within the intervals of their respective ancestors.

Theorem 1. Parenthesis theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. The intervals $[\texttt{pre}[u], \texttt{post}[u]]$ and $[\texttt{pre}[v], \texttt{post}[v]]$ are entirely disjoint:

$$\texttt{post}[v] < \texttt{pre}[u], \text{ or } \texttt{post}[u] < \texttt{pre}[v]$$

   and neither u nor v is a descendant of the other in the depth-first forest.
2. The interval $[\texttt{pre}[u], \texttt{post}[u]]$ is contained entirely within the interval $[\texttt{pre}[v], \texttt{post}[v]]$:

$$\texttt{pre}[v] < \texttt{pre}[u] < \texttt{post}[u] < \texttt{post}[v]$$

   and u is a descendant of v in a depth-first tree.
3. the interval $[\texttt{pre}[v], \texttt{post}[v]]$ is contained entirely within the interval $[\texttt{pre}[u], \texttt{post}[u]]$:

$$\texttt{pre}[u] < \texttt{pre}[v] < \texttt{post}[v] < \texttt{post}[u]$$

   and v is a descendant of u in a depth-first tree.

2.2.4 DFS Time Complexity The DFS complexity follows the same logic as the BFS complexity (sec. 2.1.2). The recursive function is called $\Theta(|V|)$ times, marking a vertex as visited only when it reaches the function. The inner loop iterates over the neighbors of each vertex, so it takes $O(|E|)$ time in total. Therefore, the overall complexity is $O(|V| + |E|)$.