Review on Graphs

Handshaking Lemma

In a party, people shook hands with each other. The sum of the number of times each person shook hands must be even. This can be proven by proving the ***handshaking lemma***: if $G = (V, E)$ is an undirected graph, then $\sum_{v \in V} \text{degree}(v) = 2|E|$.

Proof. The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. For example, in Figure 1, The degree of `node 1` is 2 and the degree of `node 2` is 3. Consider a particular, but arbitrarily chosen edge $e_k = (v_i, v_j) \in E$; it contributes a count of 1 each to $deg(v_i)$ and $deg(v_j)$ and hence a count of 2 to the total degree. Hence, $|E|$ edges contribute $2|E|$ to the total degree. Each vertex represents a person and an edge (u_i, u_j) represents u_i and u_j shook hands. The sum of hand-shakes is the sum of the vertex-degrees which is even.

Path

A ***path*** of ***length*** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. There is always a 0-length path from u to u. If there is a path p from u to u', we say that u' is ***reachable*** from u via p. If there is no path from u to u', the distance from u to u' is infinity.

Graph Representation

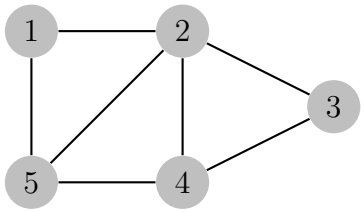
In addition to the representations shown during the lecture: adjacency lists, object-oriented variations and implicit representations, there is another way to represent a graph, called adjacency matrix.

Adjacency Matrix

For a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots |V|$ in some arbitrary order. Then the adjacency matrix representation of G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

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

This matrix can be stored as an array of arrays, and it requires $\Theta(V^2)$ memory, independent of the number of the edges in the graph. Figure 1 shows the adjacency matrix of an undirected graph. Observe the symmetry along the main diagonal of the matrix. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 1: Adjacency matrix representation of an undirected graph.

Representation Tradeoffs

Space:

- Adjacency lists uses one node per edge, and two machine words per node. So space is $\Theta(Ew)$ bits (w = word size).
- Adjacency matrix uses V^2 entries, but each entry can be just one bit. So space can be $\Theta(V^2)$ bits.

Time:

- Add an edge: both data structures are $O(1)$.
- Find if there is an edge from u to v: matrix is $O(1)$, and adjacency list must be scanned.
- Visit all neighbors of v (very common): matrix is $\Theta(V)$, and adjacency list is $O(\texttt{neighbors})$. This means BFS will take $O(V^2)$ time if we use adjacency matrix representation.
- Remove an edge: similar to find and add.

The adjacency list representation provides a compact way to represent ***sparse*** graphs – those for which $|E|$ is much less than $|V^2|$ – it is usually the method of choice. We may prefer an adjacency matrix representation, however, when the graph is ***dense*** – $|E|$ is close to $|V^2|$ – or when we need to be able to tell quickly if there is an edge connecting two given vertices.

Breadth First Search

Breadth first search (BFS) uses a queue to perform the search. A queue is a FIFO (first-in first-out) data structure. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the FIFO property of the queue and ends up being an extremely useful property. Even thougth the implementation shown in the lecture does not use a queue explicitly, it still maintains the FIFO order of visiting the nodes.

The following is the Python implementation of a queue-based BFS.

```
1  from collections import deque
2
3  class BFSResult:
4      def __init__(self):
5          self.level = {}
6          self.parent = {}
7
8  class Graph:
9      def __init__(self):
10         self.adj = {}
11
12     def add_edge(self, u, v):
13         if self.adj[u] is None:
14             self.adj[u] = []
15         self.adj[u].append(v)
16
17 def bfs(g, s):
18     '''Queue-based implementation of BFS.
19
20     Args:
21         g: a graph with adjacency list adj such that g.adj[u] is a list of u's
22            neighbors.
23         s: source.
24     '''
25     r = BFSResult()
26     r.parent = {s: None}
27     r.level = {s: 0}
28
29     queue = deque()
30     queue.append(s)
31
32     while queue:
33         u = queue.popleft()
34         for n in g.adj[u]:
35             if n not in level:
36                 r.parent[n] = u
37                 r.level[n] = r.level[u] + 1
38                 queue.append(n)
39     return r
```