Graph Representation

In lecture, we saw how we can represent a graph with an adjacency list. Here we describe another way to represent a graph, the 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.

![Figure 1: Adjacency matrix representation of an undirected graph.](image)

Representation Tradeoffs

Space:

- Adjacency lists uses one node per edge, and two machine words per node. So space is $\Theta(Ew)$ bits ($w = \text{word size}$).
- 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 \( O(\text{neighbors}) \). 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 though the implementation shown in the lecture does not use a queue explicitly, it still maintains the FIFO order of visiting the nodes.

Implementation

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

```python
from collections import deque

class BFSResult:
    def __init__(self):
        self.level = {}
        self.parent = {}

class Graph:
    def __init__(self):
        self.adj = {}

    def add_edge(self, u, v):
        if self.adj[u] is None:
            self.adj[u] = []
        self.adj[u].append(v)

    def bfs(g, s):
        '''Queue-based implementation of BFS.

        Args:
            g: a graph with adjacency list adj such that g.adj[u] is a list of u’s neighbors.
            s: source.
        '''
        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

Shortest Paths

Once we’ve run BFS on a graph, we have an array of levels, which tells us the length of a path from s to every other node in the graph. We claim that this array actually contains the length of the shortest path from s to any other node. In this way, BFS solves the single-source shortest path problem for unweighted graphs.

We can see why this is true with a very simple argument. We know that level 1 contains all the nodes which are direct neighbors of s. Level 2 contains all the nodes which are direct neighbors of level 1 nodes but haven’t already been discovered. Thus, the shortest path from s to any node in level 2 must be 2 (otherwise that node would be in level 1). Generalizing this argument, the shortest path from s to any node in level k must be k, or else that node would be in a previous level and wouldn’t have been placed in level k.

The Die Hard Problem

Say that we have a 3-gallon jug and a 2-gallon jug. We want to measure out exactly one gallon of water as quickly as possible. For this small example, it’s easy to see that we can just fill the 3-gallon jug, then pour that into the 2-gallon jug and empty the 2-gallon jug, leaving us with one gallon in the 3-gallon jug. While this example is easy enough to solve in your head, larger examples become much more complicated. In general, we can solve them with BFS.

First, we build a graph to represent this problem. Let the tuple \((a, b)\) represent the state where there are \(a\) gallons of water in the 3-gallon jug and \(b\) gallons of water in the 2-gallon jug. These states will be the nodes of our graph. Our starting state will be \((0, 0)\), and the desired ending state will be \((1, 0)\). The edges of the graph will represent possible transitions. We are allowed to perform the following actions:

1. Fill one of the jugs
2. Empty one of the jugs
3. Pour from one jug into another until either the pouring jug is empty or the receiving jug is full

The complete graph is represented in Figure 2. Next, we run BFS on the graph, which will show us all the nodes which are reachable from the starting state. In particular, we see that \((1, 0)\) is reachable, through the path \((0, 0), (3, 0), (1, 2), (1, 0)\), which corresponds to the set of steps described above.
Figure 3: The graph for our problem with a 5-gallon jug and a 3-gallon jug. Reachable states are shown in blue.

Now say that we have a 5-gallon jug and a 3-gallon jug. Again, we begin at starting state $(0, 0)$, and want to reach state $(1, 0)$. Is this possible? The graph for this problem is shown in Figure 3. We note that in this larger example there are many more unreachable states. However, we see that $(1, 0)$ is reachable through the path $(0, 0), (0, 3), (3, 0), (3, 3), (5, 1), (1, 0), (1, 1)$. In fact, we can see that $(1, 0), (2, 0), (3, 0), (4, 0), (5, 0)$ are all reachable from the starting state. This means that we can get any integer number of gallons from our two jugs.
Is this always the case? What if we had a 4-gallon jug and a 2-gallon jug? The graph is shown in Figure 4.

![Figure 4: The graph for our problem with a 4-gallon jug and a 2-gallon jug. Reachable states are shown in blue.](image)

We can see that we can never reach \((1, 0)\) or \((3, 0)\). In fact, we can only each states where both jugs have an even number of gallons. Intuitively, this is obvious. If both jugs are a multiple of two and we can only empty or fill jugs or pour them into each other, we can never create an odd number.