Breadth-First Search

Adapted from the CLRS book slides

GRAPH REPRESENTATION

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs for algorithms:

  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$ really means $O(|V| + |E|)$.

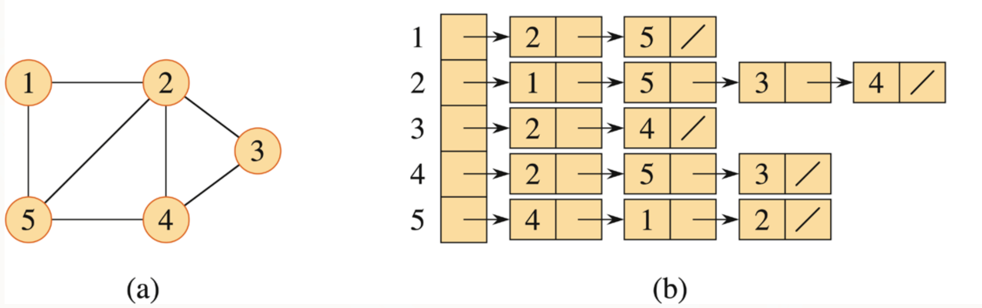
ADJACENCY LISTS

Array Adj of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

EXAMPLE



For an undirected graph:

If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

ADJACENCY MATRIX

$$|V| \times |V| \text{ matrix } A = (a_{ij})$$

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

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

BREADTH-FIRST SEARCH

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output:

- $v.d =$ distance (smallest # of edges) from s to v, for all $v \in V$.
- $v.\pi$ is v's ***predecessor*** on a shortest path (smallest # of edges) from s.
  (u, v) is last edge on shortest path $s \leadsto v$.
  Predecessor subgraph contains edges (u, v) such that $v.\pi = u$.
  The predecessor subgraph forms a tree, called the ***breadth-first tree***.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

BREADTH-FIRST SEARCH (continued)

Intuition

Breadth-first search expands the frontier between discovered and undiscovered ver tices uniformly across the breath of the frontier.

Discovers vertices in waves, starting from s.

- First visits all vertices 1 edge from s.
- From there, visits all vertices 2 edges from s.
- Etc.

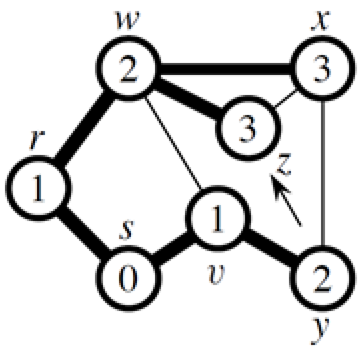
Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has visited v but has not come out of v yet.
- Q contains vertices at a distance k, and possibly some vertices at a distance $k+1$. Therefore, at any time Q contains portions of two consecutive waves.

PSEUDOCODE

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]    // search the neighbors of u
        if v.color == WHITE          // is v being discovered now?
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)            // v is now on the frontier
    u.color = BLACK                  // u is now behind the frontier
```

EXAMPLE



$Q \quad \emptyset$

- Edges drawn with heavy lines are in the predecessor subgraph.
- Dashed lines go to newly discovered vertices. They are drawn with heavy lines because they are also now in the predecessor subgraph.
- Double-outline vertices have been discovered and are in Q, waiting to be visited.
- Heavy-outline vertices have been discovered, dequeued from Q, and visited.

PSEUDOCODE

To print the vertices on a shortest path from s to v:

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print “no path from” s “to” v “exists”
else PRINT-PATH(G, s, v.π)
    print v
```