Lecture 9

• quick review of graphs

• two basic graph problems + one algorithm
  Breadth First Search (BFS)

• weighted graphs
  - a specific shortest path problem
  - a Dynamic Programming solution
How is the graph represented?

assume $V = \{1, \ldots, n \}$

(i.e., use $1 \ldots n$ to label vertices)

Adjacency matrix representation of $G$:

$n \times n$ matrix $A$

where $A[i, j] = 1$ if $(i, j) \in E$

$0$ otherwise

$\Theta(n^2)$ space - good for dense graphs!

example 1

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

example 2

$$
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 0 & 1 \\
2 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 \\
4 & 1 & 1 & 0 \\
\end{array}
$$
Adjacency-list representation of $G$:

list for each vertex $u$ containing its edges (outedges for digraphs)

Example 1

\[
\begin{array}{c}
1 \rightarrow 3 \rightarrow 4 \\
2 \rightarrow 4 \\
3 \rightarrow 1 \rightarrow 4 \\
4 \rightarrow 2 \rightarrow 3 \\
\end{array}
\]

Example 2

\[
\begin{array}{c}
1 \rightarrow 3 \\
2 \rightarrow 4 \\
3 \rightarrow 4 \\
4 \rightarrow 1 \rightarrow 2 \\
\end{array}
\]

$O(|V|+|E|)$ space

Good for "sparse" graph where $|E| \ll |V|^2$
How many edges in a graph?

- Directed: \( \leq |V|^2 \)  
- Undirected: \( \leq \binom{|V|}{2} \)  

\( \Rightarrow \) both \( \mathcal{O}(|V|^2) \)

What kinds of algorithmic questions interest us?

Example 1: Is a graph connected?

- Undirected graph - can you reach any node from any other node following edges?
- Directed graph - above is called strongly connected (other connectivity notions exist)

Example 2: How far is it from node A to B?
Breadth First Search

What does it do?

- tool for searching a graph
- can be used as a basis for many graph algorithms

Given graph $G$ and start vertex $s$

gives systematic way to explore edges of $G$ to find vertices reachable from $s$.

Main idea:
- explore unknown frontier across the breadth of frontier
- keep track of progress by coloring visited vertices
  - white: unvisited
  - black: visited + all neighbors have been visited
  - grey: visited but may have unvisited neighbors.
**BFS (G, s)**

for each \( u \in V \setminus \{s\} \)
    do color \((u)\) ← white
        \(d(u) \leftarrow \infty\)

\(\text{color}(s) \leftarrow \text{gray}\)
\(d(s) \leftarrow 0\)

\(Q \leftarrow \emptyset\)
\(\text{Enqueue} \quad (Q, s)\)

while \(Q \neq \emptyset\)
    do \(u \leftarrow \text{Dequeue} \quad (Q)\)
        for each \( v \in \text{Adj}(u) \)
            if \(\text{color} \quad (v) = \text{white}\)
                then \(\text{color}(v) \leftarrow \text{gray}\)
                \(d(v) \leftarrow d(u) + 1\)
                \(\text{Enqueue} \quad (Q, v)\)

\(\text{color} \quad (u) \leftarrow \text{black}\)

At end:
- all unreachable nodes have \(d(u) = \infty\)
- all reachable nodes have \(d(u)\) finite (and equal to distance from \(s\))
example

1. \( \text{Q: s} \)

2. \( \text{Q: a b} \)

3. \( \text{Q: b c} \)

4. \( \text{Q: c d} \)

Color code:
- \( \textcolor{blue}{\text{Blue}} \) = white
- \( \textcolor{gray}{\text{Gray}} \) = gray
- \( \textcolor{red}{\text{Red}} \) = black
Comments

- runtime:
  each vertex enqueued/dequeued at most once
  each edge scanned at most twice
  (once from each vertex)

  so \( O(\|V\| + \|E\|) \)

- final value of \( d(u) \):
  is shortest path length from \( s \) to \( u \)
  (won't prove here)

- correctness:
  BFS discovers every \( v \in V \) reachable from \( s \)
  (won't prove here either)

Summary:
Solves connectivity and distance on graphs!
Weighted Graphs

- Can assign weights to edges representing:
  - probabilities
  - cost
  - distance
  - payoff
- easy to incorporate into previous representations

Weighted Adjacency matrix

wij

Weighted Adjacency list
A shortest path problem on a special type of graph:

Ronit's morning commute

Commonwealth

Construction backups on both Beacon + Commonwealth

numbers represent time to traverse a segment

What is the best path?

try all paths? 2 choices
Let's assign variable names to distances & nodes:

\[ \begin{align*}
\text{b}_i &\text{ is times along Beacon} \\
\text{c}_i &\text{ is times along Commonwealth} \\
\text{d}_i &\text{ is times (down) from Beacon to Commonwealth} \\
\text{u}_i &\text{ is times (up) from Commonwealth to Beacon}
\end{align*} \]

"Brute Force" (consider all paths) needs \( \Omega(2^n) \) time!

Let's look at some subproblems:

- getting to \( x_1, y_1 \)
  
  only one choice \( \checkmark \)

- getting to \( x_2 \) (\( y_2 \) similar)
  
  from \( x_i \): \( b_i + b_2 \) \( \checkmark \) which is smaller?
  
  from \( y_i \): \( c_1 + u_2 \)
Key Observation:

**best** way to reach $x_i$ is **minimum** of:

- best way to reach $x_{i-1} + b_i$
- best way to reach $y_{i-1} + u_i$

(similar for $y_i$)

Why?

assume $P$ is best path to $x_i$

and assume $P$ goes through $x_{i-1}$.

Now if there is a faster way to get to $x_{i-1}$

then can use it to give faster way to $x_i$

(similar reasoning if $P$ goes through $y_{i-1}$)

Moral:

Optimal solution contains optimal solution to subproblems.

“Optimal Substructure”
A Recursive Solution:

Define subproblems:

\[ b(i) = \text{fastest time from } s \text{ to } b_i \]
\[ c(i) = \text{fastest time from } s \text{ to } c_i \]

Then ultimate goal

\[ f^* = \text{fastest time from } s \text{ to } t \]
\[ = \min \left( b(n) + b_{nt}, c(n) + c_{nt} \right) \]

and

\[ b(1) = b_1 \]
\[ c(1) = c_1 \]

and

\[ b(i) = \min \left\{ b(i-1) + b_i, c(i-1) + u_i \right\} \]
\[ c(i) = \min \left\{ c(i-1) + c_i, b(i-1) + d_i \right\} \]
Recursive Computation:

\[ b(j) \]
\[ b(j-1) \]
\[ b(j-2) \]

\[ c(j) \]
\[ c(j-2) \]
\[ c(j-2) \]

\[ b(j) \]
\[ b(j-1) \]
\[ b(j-2) \]

But they are all the same!

Only two distinct subproblems per row.

Main idea:
Write down the answers and do not recompute.
(easier to think about as building a solution "bottom up")
Fastest Way \( (b, c, d, u, n) \)

\[
\begin{align*}
    b(i) &= b_i, \\
    c(i) &= c_i, \\
    \text{for } j &< 2 \text{ to } n \\
    b(j) &\leftarrow \min \{ b(j - 1) + b_i, c(j - 1) + u_i \} \\
    c(j) &\leftarrow \min \{ c(j - 1) + c_i, b(j - 1) + d_i \}
\end{align*}
\]

Output \( \min( (b(n) + b_{n+1}), (c(n) + c_{n+1})) \)

---

Key elements:

- Optimal substructure
- Avoiding recomputations

An example of "Dynamic Programming"

(more next time... )