Why Graphs?

- Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics

Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

\[\text{Explore}(u) : \]

\begin{align*}
\text{Initialize } S &= \{u\} \\
\text{while there is an edge } (x, y) \text{ with } x \in S \text{ and } y \notin S \text{ do} \\
&\quad \text{add } y \text{ to } S
\end{align*}

DFS in Directed Graphs

\[\text{DFS}(G) : \]

Mark all nodes $u$ as unvisited

$T$ is set to $\emptyset$

\[time = 0 \]

\[\text{while there is an unvisited node } u \text{ do} \]

\[\text{DFS}(u)\]

\[\text{Output } T\]

\[\text{DFS}(u) : \]

Mark $u$ as visited

\[\text{pre}(u) = ++ time\]

\[\text{for each edge } (u, v) \text{ in } \text{Out}(u) \text{ do} \]

\[\text{if } v \text{ is not marked} \]

\[\quad \text{add edge } (u, v) \text{ to } T\]

\[\text{DFS}(v)\]

\[\text{post}(u) = ++ time\]
Node $u$ is **active** in time interval $[\text{pre}(u), \text{post}(u)]$.

**Proposition**

For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

**Directed Graph Connectivity Problems**

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $\text{rch}(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$.
- Find the strongly connected component containing node $u$, that is $\text{SCC}(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strongly connected components of $G$.

First four problems can be solve in $O(n + m)$ time by adapting BFS/DFS to directed graphs. The last one requires a clever DFS based algorithm.

**DFS Properties**

Generalizing ideas from undirected graphs:

- $\text{DFS}(u)$ outputs a directed out-tree $T$ rooted at $u$
- A vertex $v$ is in $T$ if and only if $v \in \text{rch}(u)$
- For any two vertices $x$, $y$ the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint or one is contained in the other.
- The running time of $\text{DFS}(u)$ is $O(k)$ where $k = \sum_{v \in \text{rch}(u)} |\text{Adj}(v)|$ plus the time to initialize the Mark array.
- $\text{DFS}(G)$ takes $O(m + n)$ time. Edges in $T$ form a disjoint collection of out-trees. Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.
DFS Tree

Edges of $G$ can be classified with respect to the DFS tree $T$ as:

- **Tree edges** that belong to $T$
- A **forward edge** is a non-tree edges $(x, y)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y)$.
- A **backward edge** is a non-tree edge $(x, y)$ such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.
- A **cross edge** is a non-tree edges $(x, y)$ such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.

Algorithms via DFS

$SC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}$

- Find the strongly connected component containing node $u$. That is, compute $SCC(G, u)$.

$SCC(G, u) = \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$

Hence, $SCC(G, u)$ can be computed with two DFSes, one in $G$ and the other in $G^{rev}$. Total $O(n + m)$ time.

Linear Time Algorithm

...for computing the strong connected components in $G$

```
    do DFS($G^{rev}$) and sort vertices in decreasing post order.
    Mark all nodes as unvisited
    for each $u$ in the computed order do
        if $u$ is not visited then
            DFS($u$)
            Let $S_u$ be the nodes reached by $u$
            Output $S_u$ as a strong connected component
            Remove $S_u$ from $G$
```

Analysis

Running time is $O(n + m)$. (Exercise)

Example: Makefile

BFS with Distances

$\text{BFS}(s)$

Mark all vertices as unvisited and for each $v$ set $\text{dist}(v) = \infty$

Initialize search tree $T$ to be empty
Mark vertex $s$ as visited and set $\text{dist}(s) = 0$
set $Q$ to be the empty queue
enq($s$)
while $Q$ is nonempty do
    $u = \text{deq}(Q)$
    for each vertex $v \in \text{Adj}(u)$ do
        if $v$ is not visited do
            add edge $(u, v)$ to $T$
            Mark $v$ as visited, $\text{enq}(v)$
            and set $\text{dist}(v) = \text{dist}(u) + 1$
```

Proposition

$\text{BFS}(s)$ runs in $O(n + m)$ time.
BFS with Layers

**BFSLayers**($s$):
Mark all vertices as unvisited and initialize $T$ to be empty
Mark $s$ as visited and set $L_0 = \{s\}$

$i = 0$
while $L_i$ is not empty do
initialize $L_{i+1}$ to be an empty list
for each $u$ in $L_i$ do
for each edge $(u, v) \in \text{Adj}(u)$ do
if $v$ is not visited
mark $v$ as visited
add $(u, v)$ to tree $T$
add $v$ to $L_{i+1}$
i = $i + 1$

Running time: $O(n + m)$

Checking if a graph is bipartite...

**Linear time algorithm**

**Corollary**
There is an $O(n + m)$ time algorithm to check if $G$ is bipartite and output an odd cycle if it is not.

Dijkstra’s Algorithm

Initialize for each node $v$, $\text{dist}(s, v) = \infty$
Initialize $S = \{s\}$, $\text{dist}(s, s) = 0$
for $i = 1$ to $|V|$ do
Let $v$ be such that $\text{dist}(s, v) = \min_{u \in V \setminus S} \text{dist}(s, u)$
$S = S \cup \{v\}$
for each $u$ in $\text{Adj}(v)$ do
$\text{dist}(s, u) = \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u))$

- Using Fibonacci heaps. Running time: $O(m + n \log n)$.
- Can compute shortest path tree.

Single-Source Shortest Paths with Negative Edge Lengths

**Input:** A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
- Given nodes $s, t$ find shortest path from $s$ to $t$.
- Given node $s$ find shortest path from $s$ to all other nodes.
Negative Length Cycles

**Definition**
A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.

A Generic Shortest Path Algorithm

Dijkstra’s algorithm does not work with negative edges.

Relax$(e = (u, v))$
if $(d(s, v) > d(s, u) + ℓ(u, v))$ then
$d(s, v) = d(s, u) + ℓ(u, v)$

GenericShortestPathAlg:
$d(s, s) = 0$
for each node $u \neq s$ do
$d(s, u) = \infty$
while there is a tense edge do
Pick a tense edge $e$
Relax$(e)$
Output $d(s, u)$ values

Bellman-Ford to detect Negative Cycles

for each $u \in V$ do
$d(s, u) = \infty$
$d(s, s) = 0$
for $i = 1$ to $|V| - 1$ do
for each edge $e = (u, v)$ do
Relax$(e)$
for each edge $e = (u, v)$ do
if $e = (u, v)$ is tense then
Stop and output that $s$ can reach a negative length cycle
Output for each $u \in V$: $d(s, u)$

Total running time: $O(mn)$.
Can detect negative cycle reachable from $s$.
Appropriate construction - detect any negative cycle in a graph.

Shortest paths in DAGs

Algorithm for DAGs:

ShorestPathInDAG$(G, s)$:
$s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of $G$
for $i = 1$ to $n$ do
$d(s, v_i) = \infty$
$d(s, s) = 0$
for $i = 1$ to $n - 1$ do
for each edge $e$ in $\text{Adj}(v_i)$ do
Relax$(e)$
return $d(s, \cdot)$ values computed

Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find longest paths in a DAG.
Reduction

Reduction problem $A$ to problem $B$:
- Algorithm for $A$ uses algorithm for $B$ as a black box.
- Example: Uniqueness (or distinct element) to sorting.

Recursion

Recursion is a very powerful and fundamental technique.
- Basis for several other methods.
  - Divide and conquer.
  - Dynamic programming.
  - Enumeration and branch and bound etc.
  - Some classes of greedy algorithms.
- Recurrences arise in analysis.

Examples seen:
- Recursion: Tower of Hanoi, Selection sort, Quick Sort.
- Divide & Conquer:
  - Merge sort.
  - Multiplying large numbers.

Solving recurrences using recursion trees

An illustrated example: Merge sort...

\[ n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \ldots \rightarrow \log n \]

Work in each node

Recurrence tree:

- $T(n) = \Theta(n \log n)$

Solving recurrences

The other “technique” - guess and verify

- Guess solution to recurrence.
- Verify it via induction.

Solved in class:
- $T(n) = 2T(n/2) + n/ \log n$.
- $T(n) = T(\sqrt{n}) + 1$.
- $T(n) = nT(\sqrt{n}) + n$.
- $T(n) = T(n/4) + T(3n/4) + n$.
Closest Pair - the problem

Input  Given a set $S$ of $n$ points on the plane
Goal  Find $p, q \in S$ such that $d(p, q)$ is minimum

Algorithm:
One can compute closest pair points in the plane in $O(n \log n)$ time using divide and conquer.

Median selection

Problem
Given list $L$ of $n$ numbers, and a number $k$ find $k$th smallest number in $n$.

- Quick Sort can be modified to solve it (but worst case running time is quadratic (if lucky linear time).
- Seen divide & conquer algorithm... Involved, but linear running time.

Recursive algorithm for Selection

A feast for recursion

```
select(A, j):
    if n <= 10 then
        Compute jth smallest element in A using brute force.
        Form lists $L_1, L_2, \ldots, L_{\lfloor n/5 \rfloor}$ where $L_i = \{A[5i - 4], \ldots, A[5i]\}$
        Find median $b_i$ of each $L_i$ using brute-force
        $B$ is the array of $b_1, b_2, \ldots, b_{\lfloor n/5 \rfloor}$.
        $b = \text{select}(B, \lfloor n/10 \rfloor)$
        Partition $A$ into $A_{\text{less or equal}}$ and $A_{\text{greater}}$ using $b$ as pivot
        if $|A_{\text{less or equal}}| = j$ then
            return $b$
        if $|A_{\text{less or equal}}| > j$ then
            return select($A_{\text{less or equal}}$, $j$)
        else
            return select($A_{\text{greater}}$, $j - |A_{\text{less or equal}}|$)
```