Review session

Lecture 666
February 24, 2015
Why Graphs?

1. Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.

2. Fundamental objects in Computer Science, Optimization, Combinatorics

3. Many important and useful optimization problems are graph problems

4. Graph theory: elegant, fun and deep mathematics
Basic Graph Search

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

 explores $u$:

 Initialize $S = \{u\}$

 while there is an edge $(x, y)$ with $x \in S$ and $y \notin S$ do

 add $y$ to $S$
DFS in Directed Graphs

**DFS\((G)\)**

Mark all nodes \(u\) as unvisited
\[ T \text{ is set to } \emptyset \]
\[ \textit{time} = 0 \]

while there is an unvisited node \(u\) do

DFS\((u)\)

Output \(T\)

**DFS\((u)\)**

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

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

if \(v\) is not marked

add edge \((u, v)\) to \(T\)

DFS\((v)\)

post\((u)\) = \text{++ time}
**pre and post numbers**

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.
Connectivity and Strong Connected Components

**Definition**

Given a directed graph $G$, $u$ is strongly connected to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$. 

![Diagram](https://via.placeholder.com/150)
Directed Graph Connectivity Problems

1. Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
2. Given $G$ and $u$, compute $rch(u)$.
3. Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in rch(v)$.
4. Find the strongly connected component containing node $u$, that is $SCC(u)$.
5. Is $G$ strongly connected (a single strong component)?
6. 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:

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

1. **Tree edges** that belong to $T$

2. A **forward edge** is a non-tree edges $(x, y)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

3. A **backward edge** is a non-tree edge $(x, y)$ such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.

4. 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.
SC(G, u) = \{ v \mid u \text{ is strongly connected to } v \}

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

SCC(G, u) = rch(G, u) \cap 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.
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 \( SCC(G, u) \).

\[ SCC(G, u) = rch(G, u) \cap rch(G^{rev}, u) \]

Hence, \( SCC(G, u) \) can be computed with two \text{DFS}es, 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 $\text{DFS}(G^{\text{rev}})$ 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
    $\text{DFS}(u)$
  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

**BFS**(s)

Mark all vertices as unvisited and for each \( v \) set \( \text{dist}(v) = 1 \)

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, **enq**(v)

and set \( \text{dist}(v) = \text{dist}(u) + 1 \)

---

**Proposition**

**BFS**\((s)\) runs in \( O(n + m) \) time.
BFS with Layers

\textbf{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: $O(n + m)$
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 \( \text{dist}(s, v) = \min_{u \in V - S} \text{dist}(s, u) \)

\( S = S \cup \{v\} \)

for each \( u \) in \( \text{Adj}(v) \) do

\( \text{dist}(s, u) = \min \left( \text{dist}(s, u), \text{dist}(s, v) + \ell(v, u) \right) \)

1. Using Fibonacci heaps. Running time: \( O(m + n \log n) \).
2. Can compute shortest path tree.
Single-Source Shortest Paths with Negative Edge Lengths

**Single-Source Shortest Path Problems**

**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.
Definition

A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.
Dijkstra’s algorithm does not work with negative edges.

\[ \text{Relax}(e = (u, v)) \]
\[ \text{if } (d(s, v) > d(s, u) + \ell(u, v)) \text{ then} \]
\[ d(s, v) = d(s, u) + \ell(u, v) \]

\textbf{GenericShortestPathAlgo}:
\[ d(s, s) = 0 \]
\[ \text{for each node } u \neq s \text{ do} \]
\[ d(s, u) = \infty \]

\textbf{while} there is a tense edge \textbf{do}
\[ \text{Pick a tense edge } e \]
\[ \text{Relax}(e) \]

\textbf{Output} \[ d(s, u) \] values
Bellman-Ford to detect Negative Cycles

\[
\begin{align*}
\text{for each } u & \in V \text{ do} \\
& d(s, u) = \infty \\
& d(s, s) = 0 \\
\text{for } i & = 1 \text{ to } |V| - 1 \text{ do} \\
& \text{for each edge } e = (u, v) \text{ do} \\
& \quad \text{Relax}(e) \\
\text{for each edge } e & = (u, v) \text{ do} \\
& \quad \text{if } e = (u, v) \text{ is tense then} \\
& \quad \quad \text{Stop and output that } s \text{ can reach} \\
& \quad \quad \text{a negative length cycle} \\
& \quad \text{Output for each } u \in V: \quad d(s, u) \\
\end{align*}
\]

1. Total running time: \(O(mn)\).
2. Can detect negative cycle reachable from \(s\).
3. Appropriate construction - detect any negative cycle in a graph.
Shortest paths in **DAGs**

**Algorithm for DAGs**

### ShortestPathInDAG($G$, $s$):

1. $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of $G$.
2. for $i = 1$ to $n$ do
   - $d(s, v_i) = \infty$
   - $d(s, s) = 0$
3. for $i = 1$ to $n - 1$ do
   - for each edge $e$ in Adj($v_i$) do
     - Relax($e$)
4. 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

Reducing problem $A$ to problem $B$:

1. Algorithm for $A$ uses algorithm for $B$ as a black box.
2. Example: Uniqueness (or distinct element) to sorting.
Recursion

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

Examples seen:

1. Recursion: Tower of Hanoi, Selection sort, Quick Sort.
2. Divide & Conquer:
   1. Merge sort.
   2. Multiplying large numbers.
Solving recurrences using recursion trees

An illustrated example: Merge sort...
Solving recurrences using recursion trees

An illustrated example: Merge sort...

Work in each node
Solving recurrences using recursion trees

An illustrated example: Merge sort...

\[ n \quad cn \]
\[ \frac{n}{2} \quad \frac{cn}{2} \]
\[ \frac{n}{4} \quad \frac{cn}{4} \quad \frac{n}{4} \quad \frac{cn}{4} \quad \frac{n}{4} \quad \frac{cn}{4} \]

Work in each node
Solving recurrences using recursion trees
An illustrated example: Merge sort...

\[
\begin{align*}
\log n \left\{ \begin{array}{c}
\frac{cn}{2} + \frac{cn}{2} = cn \\
\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn \\
\vdots \\
= cn
\end{array} \right. 
\end{align*}
\]
Solving recurrences using recursion trees

An illustrated example: Merge sort...

\[
\log n \begin{cases}
\frac{c_n}{2} + \frac{c_n}{2} \\
\frac{c_n}{4} + \frac{c_n}{4} + \frac{c_n}{4} + \frac{c_n}{4} \\
\vdots
\end{cases} = c_n + c_n + c_n + c_n = c_n \log n = O(n \log n)
\]
Solving recurrences

The other “technique” - guess and verify

1. Guess solution to recurrence.
2. Verify it via induction.

Solved in class:

1. \( T(n) = 2T(n/2) + n/\log n \).
2. \( T(n) = T(\sqrt{n}) + 1 \).
3. \( T(n) = \sqrt{n}T(\sqrt{n}) + n \).
4. \( T(n) = T(n/4) + T(3n/4) + n \).
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.
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.
Problem
Given list \( L \) of \( n \) numbers, and a number \( k \) find \( k \)th smallest number in \( n \).

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

A feast for recursion

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

Seen some simple recursive algorithms:
1. Binary search.
2. Fast exponentiation.
3. Fibonacci numbers.