Review session

Lecture 99

September 22, 2011

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 $u \in V$:

```
Explore(u):
    Initialize S = {u}
    while there is an edge (x, y) with x ∈ S and y ∉ S do
        add y to S
```


DFS in Directed Graphs

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    while there is an unvisited node u do
        DFS(u)

    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u, v) in Out(u) do
        if v is not marked
            add edge (u, v) to T
            DFS(v)
    post(u) = ++time
```

pre and post numbers

Node u is ***active*** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

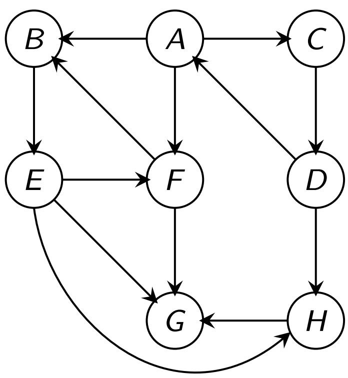
Proposition

For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ 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 $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.



Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\mathsf{rch}(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in \mathsf{rch}(v)$.
- Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
- 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:

- $DFS(u)$ outputs a directed out-tree T rooted at u
- A vertex v is in T if and only if $v \in \mathsf{rch}(u)$
- For any two vertices x, y the intervals $[\mathrm{pre}(x), \mathrm{post}(x)]$ and $[\mathrm{pre}(y), \mathrm{post}(y)]$ are either disjoint are one is contained in the other.
- The running time of $DFS(u)$ is $O(k)$ where $k = \sum_{v \in \mathsf{rch}(u)} |Adj(v)|$ plus the time to initialize the Mark array.
- **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.

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 $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
- A ***backward edge*** is a non-tree edge (x, y) such that $\mathrm{pre}(y) < \mathrm{pre}(x) < \mathrm{post}(x) < \mathrm{post}(y)$.
- A ***cross edge*** is a non-tree edges (x, y) such that the intervals $[\mathrm{pre}(x), \mathrm{post}(x)]$ and $[\mathrm{pre}(y), \mathrm{post}(y)]$ 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 $\mathrm{SCC}(G, u)$.

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

Hence, $\mathrm{SCC}(G, u)$ can be computed with two **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 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

```
BFS(s)
    Mark all vertices as unvisited and for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

Proposition

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 $(u, v) \in \mathrm{Adj}(u)$ **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, dist(s, v) = ∞
Initialize S = {s}, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−S} dist(s, u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

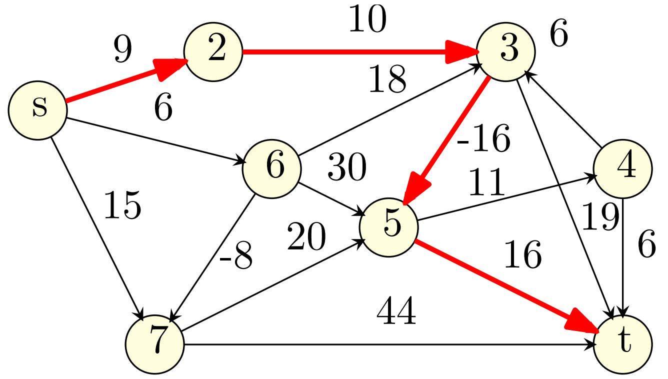
- Using Fibonacci heaps. Running time: $O(m + n \log n)$.
- Can compute shortest path tree.

Single-Source Shortest Paths with Negative Edge Lengths

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.

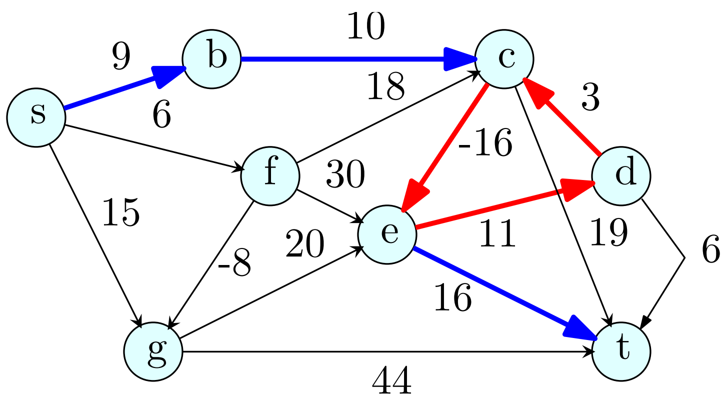
- Given nodes s, t find shortest path from s to t.
- Given node s find shortest path from s to all other nodes.



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 $(d(s, v) > d(s, u) + \ell(u, v))$ **then**
    $d(s, v) = d(s, u) + \ell(u, v)$

$d(s, s) = 0$
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 ∈ V do
    d(s, u) = ∞
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 ∈ V: d(s, u)
```

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

Shortest paths in DAGs

Algorithm for DAGs

```
ShortestPathInDAG(G, s):
    s = v_1, v_2, v_{i+1}, ..., v_n be a topological sort of G
    for i = 1 to n do
        d(s, v_i) = ∞
    d(s, s) = 0

    for i = 1 to n − 1 do
        for each edge e in Adj(v_i) do
            Relax(e)

    return d(s, ·) 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 $\boldsymbol{A}$ to problem $\boldsymbol{B}$:

- Algorithm for $\boldsymbol{A}$ uses algorithm for $\boldsymbol{B}$ as a *black box*.
- Example: Uniqueness (or distinct element) to sorting.

Recursion

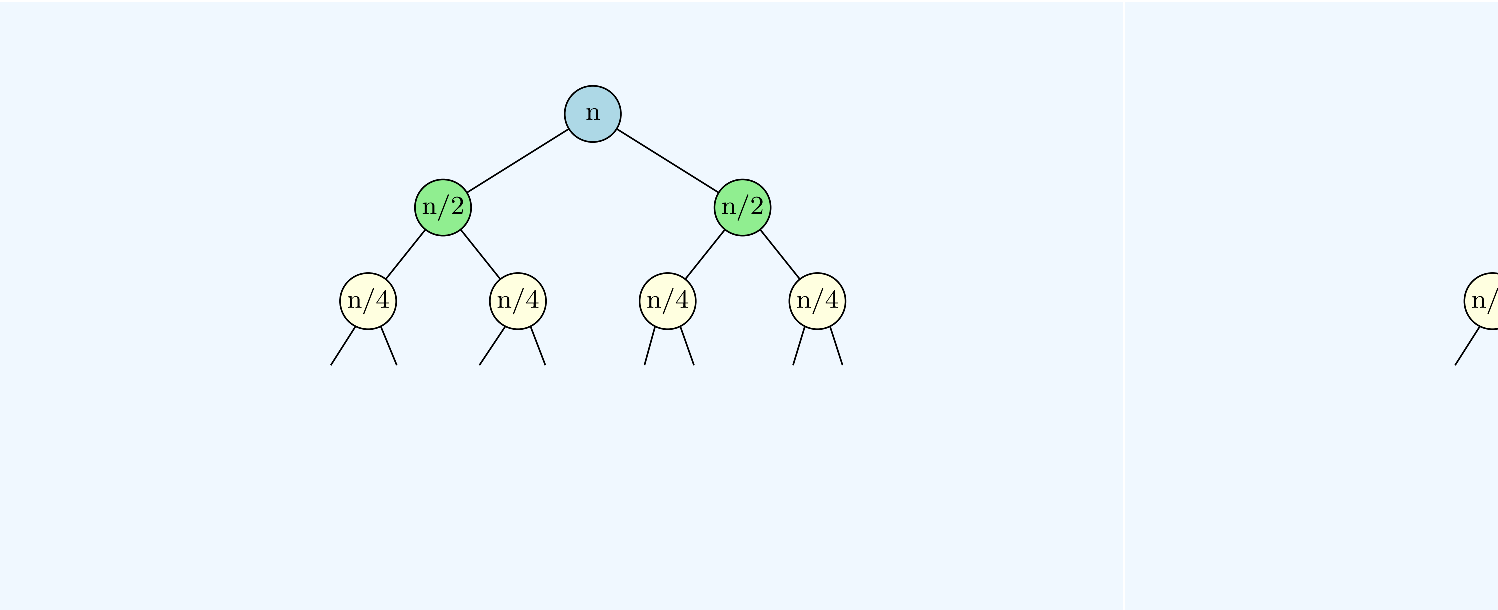
- Recursion is a very powerful and fundamental technique.
- Basis for several other methods.
  - Divide and conquer.
  - Dynamic programming.
  - Enumeration and branch and bound etc.
  - Some classes of greedy algorithms.
- Recurrences arise in analysis.

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...



Solving recurrences

The other "technique" - guess and verify

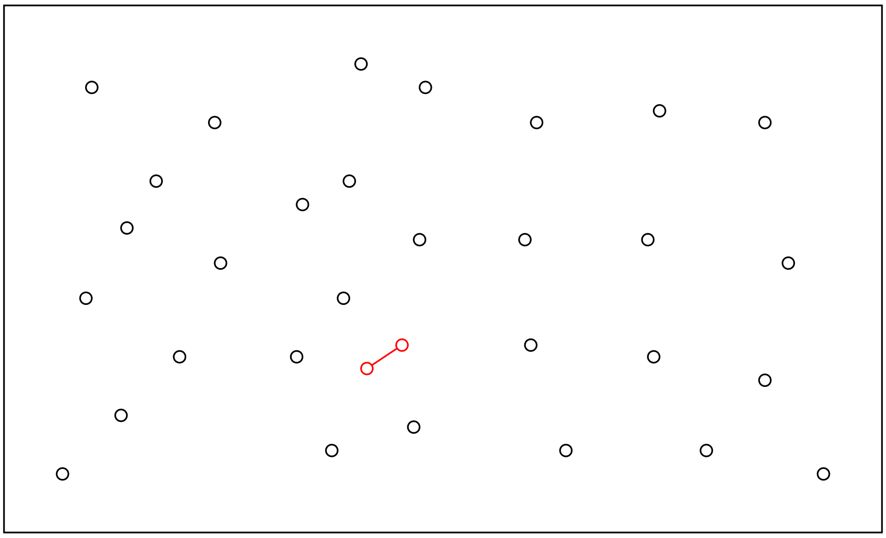
- Guess solution to recurrence.
- Verify it via induction.

Solved in class:

- $T(n) = 2T(n/2) + n/\log n$.
- $T(n) = T(\sqrt{n}) + 1$.
- $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
- $T(n) = T(n/4) + T(3n/4) + n$

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 kth 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):
  $n = |A|$
  if $n \leq 10$ **then**
    Compute jth smallest element in A using brute force.
  Form lists $L_1, L_2, \ldots, L_{\lceil n/5 \rceil}$ 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 $b_1, b_2, \ldots, b_{\lceil n/5 \rceil}$.
  $b =$ select(B, $\lceil n/10 \rceil$)
  Partition A into $A_{\texttt{less or equal}}$ and $A_{\texttt{greater}}$ using b as pivot
  if $|A_{\texttt{less or equal}}| = j$ **then**
    return b
  if $|A_{\texttt{less or equal}}| > j$) **then**
    return select($A_{\texttt{less or equal}}$, j)
  else
    return select($A_{\texttt{greater}}$, $j - |A_{\texttt{less or equal}}|$)

Back to Recursion

Seen some simple recursive algorithms:

- Binary search.
- Fast exponentiation.
- Fibonacci numbers.
- Maximum weight independent set.