CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 20: November 3, 2016

Tree Traversal and Graph Search

- Traversal and search refer to the systematic visiting of the nodes of a tree or a graph, performing certain operations at each node.
- In the last lecture we showed that we can perform tree traversal in $O(n)$ time, n being the number of nodes in the tree.
- In this lecture we will focus on searching through a general graph.
- Let $G(V, E)$ be a given undirected graph that we are interested in searching. There are several ways of searching. Two popular methods are Breadth-First Search (BFS) and Depth-First Search (DFS).
- Recall that G can be represented as adjacency lists or an adjacency matrix. Let $V = \{1, 2, \ldots, n\}$.

  The adjacency lists representation of G is an array $A[1 : n]$ of lists. $A[i]$ is a list of all the neighbors of the node i, $1 \leq i \leq n$.

  The adjacency matrix representation of G is a $n \times n$ matrix A such that $A[i, j] = 1$ if there is an edge from the node i to node j in G; and $A[i, j] = 0$ otherwise.
- BFS starts from a node, say, u. The node u is visited first. Nodes that are at a distance of 1 from u are visited next; Nodes that are at a distance of 2 from u are visited next; and so on.

Depth First Search (DFS)

- In DFS we start from a node, say, u and visit a neibhor v of u that has not been visited before; From v we visit a neighbor w of v that has not been visited before, and so on, until we reach a node x such that all the neighbors of x have already been visited. When this happens we backtrack to the node y that was visited before x and start the search from y, etc. The search terminates when we backtrack to the start node u.
- A pseudocode for DFS follows. To begin with, each entry in the array $visited[1 : n]$ is zero.

```
DFS(u)
    1) visited[u] = 1;
    2) for each w ∈ Adj(u) do
```

3) **if** !$visited[w]$ **then** DFS(w);

Run Time Analysis: Note that, for any node $u \in V$, line 3 is executed d_u times where d_u is the degree of u. Lines 1 and 2 are executed for every node u in V. Thus the run time of this algorithm is $O(|V| + \sum_{u \in V} d_u) = O(|V| + |E|)$. This is a linear time algorithm.

The case of multiple components

- The input graph may not be connected. Recall that an undirected graph is said to be connected if there is a path from every node to every other node in the graph.
- If the graph is not connected, then it has more than one *connected components*. A connected component of a graph is a maximal subgraph of the graph that is connected.
- When the input graph has more than one connected components we can modify the algorithm DFS to get the following algorithm DFST:

```
1) for i = 1 to n do
2)       visited[i] = 0;
3) DFST(G(V, E))
4)       for i = 1 to n do
5)            if !visited[i] then DFS(i);
```

- **Run Time:** When DFS is called on any node i, all the nodes in the connected component that i belongs to will be visited. Let the number of connected components in G be c. Let the number of nodes and edges in connected component q be $|V_q|$ and $|E_q|$, respectively, for $1 \leq q \leq c$. If the node i belongs to connected component q, then, the time spent by DFS(i) will be $O(|V_q| + |E_q|)$.

  Thus the total run time of the algorithm will be $O\left(\sum_{q=1}^{c}(|V_q| + |E_q|)\right) = O(|V| + |E|)$.

Hints on Problem 7 in Homework 2

- Note that the adjacency matrix A has information about paths of length 1 in the graph. Specifically, $A[i, j] = 1$ iff there is an edge from the node i to node j.
- Now consider the matrix A^2. $A^2[i, j]$ will be 1 only if there is a k such that $A[i, k] = 1$ and $A[k, j] = 1$, i.e., if there is a path from i to j of length 2.
- Similarly, we can prove by induction that $A^k[i, j] = 1$ only if there is a path from node i to node j of length k.

- Therefore, it follows that $A^* = I + A + A^2 + \cdots + A^{n-1}$.
- Using the binomial theorem, we can show that $I + A + A^2 + \cdots + A^{n-1} = (I + A)^{n-1}$.
- Let a be a real number and n be an integer. We can compute a^n using $n-1$ multiplications.
- In fact we can compute a^n using only $O(\log n)$ multiplications. Consider the case when $n = 2^q$ for some integer q. Then we can repeatedly square elements starting from a to get the following sequence: $a, a^2, a^4, \ldots, a^{2^q}$. Clearly, the computation of a^{2^q} takes only $O(q) = O(\log n)$ multiplications.
- Even when n is not an integral power of two we can compute a^n using $O(\log n)$ multiplications. Express n in binary form as: $n = \sum_{i=0}^{q} b_i 2^i$ where each b_i is a bit. Note that $q = O(\log n)$.
$$a^n = a^{\sum_{i=0}^{q} b_i 2^i} = \Pi_{(0 \le i \le q) \text{ and } b_i = 1} a^{2^i}.$$
- The above equation suggests the following algorithm: 1) Compute the sequence: $a, a^2, \ldots, a^{2^q}$ in $O(q)$ time; and 2) Multiply the appropriate powers of a from the above sequence. This takes $O(q)$ time as well.

  The total run time is $O(q) = O \log n)$.

Parallel Algorithms

- The idea of parallel computing is to employ multiple processors to solve a problem.
- Let π be any problem for which the best known sequential algorithm takes S time. Let P be the number of processors used and let T be the parallel run time.

  Fact: $T \ge \frac{S}{P}$.

  Proof: by contradiction. Assume to the contrary that there is a parallel algorithm that takes $< \frac{S}{P}$ time.

  We can sequentially simulate each step of the parallel algorithm in $\le P$ steps. This means that we can sequentially simulate the entire parallel algorithm in a total of $\le PT < S$ time! This is a contradiction to the fact that S is the best known sequential run time for solving π. $\square$