Asymptotic and Graphs

Julius January 17, 2024

1 Asymptotic Notation

1.1 Upper Bound, Lower Bound, Tight Bound

Def. $O(g(n)) = \{f(n) | \exists C, n_0 \in \mathbb{R}^+, s.t. \forall n \geq n_0, 0 \leq f(n) \leq Cg(n)\}$

- Every Quadratic function of n is in $O(n^2)$
- Every Linear function of n is in $O(n^2)$
- Every Cubic function of n is not in $O(n^2)$

Def. $\Omega(g(n)) = \{f(n) | \exists C, n_0 \in \mathbb{R}^+ s.t. \forall n \geq n_0, f(n) \geq Cg(n)\}$

- Every Quadratic function of n is in $\Omega(n^2)$
- Every Linear function of n is not in $\Omega(n^2)$
- Every Cubic function of n is in $\Omega(n^2)$

Def. $\Theta(g(n)) = \{f(n) | \exists C_1, C_2, n_0 \in \mathbb{R}^+ s.t. \forall n \geq n_0, 0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)\}$

- Every Quadratic function of n is in $\Theta(n^2)$
- Every Linear function of n is not in $\Theta(n^2)$
- Every Cubic function of n is not in $\Theta(n^2)$

Def. $f(n) = o(g(n)) \leftrightarrow f(n) = O(g(n)), f(n) \neq \Theta(g(n))$ $\quad f(n) = \omega(g(n)) \leftrightarrow f(n) = \Omega(g(n)), f(n) \neq \Theta(g(n))$

	Worst Case	Best Case
Linear Search	$O(n), \Omega(n), \Theta(n)$	$O(1), \Omega(1), \Theta(1)$
Binary Search	$O(lgn), \Omega(lgn), \Theta(lgn)$	$O(1), \Omega(1), \Theta(1)$
Insertion Sort	$O(n^2), \Omega(n^2), \Theta(n^2)$	$O(n), \Omega(n), \Theta(n)$
Merge Sort	$O(nlgn), \Omega(nlgn), \Theta(nlgn)$	$O(nlgn), \Omega(nlgn), \Theta(nlgn)$

Comparing growth of functions: $f_1(n) = 3^n n^2 lg^5 n$ $\quad f_2(n) = 2^n n^8 lgn$. Which function grows faster?
Because exponential component is fastest growing, we can ignore the polynomial and logarithmic components.
Two algorithms A and B that solve the same problem have the following worst-case runtime complexities:
Algorithm A: $O(3^n n^2 lg^5 n)$. Algorithm B: $O(2^n n^8 lgn)$.
Which algorithm runs faster? It's hard to define because the O is just a upper bound, and it's hard to define the average case. There are some cases we need to create **hybrid algorithms** to solve the problem.

1.2 Average Case Analysis

Why don't we perform average case performance analysis?
Given different environments, the average case performance of an algorithm is not a constant. It really depends on what data set you are using the algorithm to.

2 Graphs

2.1 BFS and DFS

What are we searching for?
1. To find out if there is a path from node A to node B
2. To find out all nodes that can be reached from A

We turn to use Adjacency List $int\ h[N], e[M], w[M], ne[M], idx$ to represent the sparse graph and Adjacency Matrix $int\ g[N][N]$ to represent dense graph. We use **BFS** with **queue and distance array** to find the shortest path, while we use **DFS** with **label array, recursion and backtrace**.

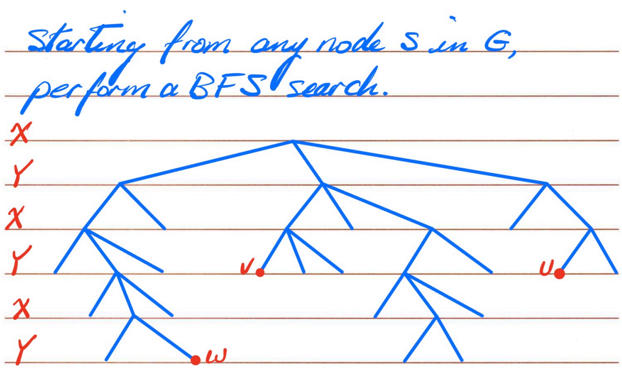
```
// Find the Distance from node u to node n
int h[N], e[M], ne[M], idx; // adjacency list for sparse graph
int d[N]; memset(d, -1, sizeof d); // Initialize -1 for all v in V to represent distance
int n, q[N], hh = 0, tt = -1; // array to represent queue
int bfs(int u){
    q[++tt] = u;
    d[u] = 0;
    while(hh <= tt){
        int s = q[hh++];
        for(int i = h[s]; i != -1; i = ne[i]){
            int j = e[i]; // j is the node connected to s
            if(d[j] == -1){
                d[j] = d[s] + 1;
                st[j] = true; // mark j as visited
                q[++tt] = j;
            }
        }
    }
    return d[n]; // return the distance from node u to node n
}

// Find the Full Permutation of 1 to n
int n, path[N];
bool st[N];
void dfs(int u){
    if(u == n){
        for(int i = 0; i < n; i++) printf("%d ", path[i]);
        puts("");
    }else{
        for(int i = 1; i <= n; i ++){ // try all possible number nodes
            if(!st[i]){
                path[u] = i;
                st[i] = true;
                dfs(u+1);
                st[i] = false;
            }
        }
    }
}
```

2.2 Bipartite

Given an undirected connected graph G, how do we determine if G is bipartite, where you could split nodes into two sets and all edges go between two sets?
A graph is bipartite $\iff$ There is no odd cycle in a graph $\iff$ There is no contradiction during staining

BFS Tree for Bipartite:

Run BFS start from any node, say s. Label each node X or Y depending on whether they appear at an odd or even level in BFS tree. O(|V| + |E|)
Then, go through all edges and exam all the lebels at the two end of the edge. If all edge ends have different labels, then the graph is bipartite. Otherwise, it's not.



Facts: Not possible to have an edge between (v, w) in G, since they are more than one level apart in BFS tree.
Facts: Edges in G that end up with the same set X or Y, must have both ends at the same level in BFS tree;

```
int h[N], e[M], ne[M], idx; // Adjacency List
int color[N]; // 0: without color 1: balck 2: red
bool dfs(int x, int c){ // return true if no contradiction during staining x with color c
    color[x] = c;
    for(int i = h[x]; i != -1; i = ne[i]){ // color all the nodes connected with x
        int j = e[i];
        if(!color[j]){ // only when without color, call dfs
            if(!dfs(j, 3 - c)) return false;
        }else if(color[j] == c){ // with color but conflict
            return false;
        }
    }
    return true;
}
void add(int a, int b){ e[idx] = b, ne[idx] = h[a], h[a] = idx ++;}
int main(){
    scanf("%d%d", &n, &m); // n: number of nodes, m: number of edges
    memset(h, -1, sizeof h);
    while (m -- ){
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b), add(b, a); // undirected graph
    }
    bool flag = true; // true: no conflict
    for(int i = 1; i <= n; i ++){
        if(!color[i]){ // only when without color, color black
            if(!dfs(i, 1)){
                flag = false;
                break;
            }
        }
    }
}
```

Weekly connected if for any pair of nodes (v, u), there is a path by ignoring edge directions.
Connected Graph if for any pair of nodes (v, u), there is a either a directed path from v to u or from u to v.
Strongly connected if for any pair of nodes (v, u), there is a directed path from v to u and a path from u to v.
Not connected if a directed graph is not connected in any of the above 3 ways.

How can we determine if a directed graph G is strongly connected?

Brute Force Solution $O(n(n+m)) = O(n^2+nm)$: Run BFS/DFS from every node in G. If we can reach all nodes from every node, then the graph is strongly connected.

Better Solution $O(n+m)$: First run BFS/DFS from an arbitrary node S see if it can reach all other nodes in G. Then we get the G^T (reverse all the edges in $O(n+m)$) and run BFS/DFS from node S again to see if it can reach all other nodes in G^T. If both BFS/DFS can reach all nodes, then the graph is strongly connected, because all the nodes are mutually reachable through node S.

2.3 Topological Order

Topological Order is the linear ordering of vertices of a graph, such that for every directed edge (u,v), vertex u comes before v in the ordering.
How to find a topological order in a Directed Acyclic Graph?

In-degree of a node is the number of edges pointing to it.

Out-degree of a node is the number of edges pointing from it.

Kahn's Algorithm $O(n+m)$: First find a node with in-degree 0, then remove it and all edges pointing from it. Decrease the in-degree of the end nodes of the removed edges by 1. Repeat this process until all nodes are removed. The order of removal is the topological order.

```
int h[N], e[N], ne[N], idx; // directed graph don't need 2 * N edges
int q[N], d[N], hh, tt=-1; // d[N] is the indegree of all nodes
void toposort(){
    for (int i = 1; i <= n; i ++ ){ // find all nodes in d[N] with indegree 0
        if(!d[i]) q[++tt] = i;
    }
    while(hh <= tt){
        int t = q[hh++];
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            d[j] --; // delete the edge t->j
            if(!d[j]) q[++tt] = j;
        }
    }
    if(tt == n-1){ // if all nodes are in the queue, then there is a topological order
```

```
        for(int i = 0; i < n; i++) printf("%d ", q[i]);
    }else{
        puts("-1");
    }
}
int main(){
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m -- ){
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b);
        d[b] ++; // remember to add the indegree of node b
    }
    toposort();
    return 0;
}
```

3 Discussion

3.1 Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if f(n) = O(g(n)).

$$logn^n, n^2, n^{logn}, nlog(logn), 2^{logn}, log^2n, n^{\sqrt{2}}$$

$$log^2n \leq 2^{logn} = n \leq nlog(logn) \leq logn^n = nlogn \leq n^{\sqrt{2}} \leq n^2 \leq n^{logn}$$

3.2 Suppose that f(n) and g(n) are two positive non-decreasing functions such that f(n) = O(g(n)). Is it true that $2^{f(n)} = O(2^{g(n)})$?

False. $f(n) = 2n, g(n) = n$, then $2^{f(n)} = 2^{2n}, 2^{g(n)} = 2^n$, $(2^n)^2 \neq O(2^n)$

3.3 Find an upper bound Big O on the worst case run time of the following code segment. Carefully examine to see if this is a tight upper bound Big Θ

```
void bigOh1(int[] L, int n)
while(n > 0)
    find max(L, n); // finds the max in L[0..n-1]
    n = n/4;
```

There are log_4n iterations, and each iteration takes $O(n)$ time. So the upper bound is $O(nlogn)$. While for tight upper bound, we have to add up the number of operations. Let's say the first iteration cost c*n, then c*n/4, ..., we get $\sum_{i=0}^{\infty} \frac{cn}{4^i} = 4cn/3$, we can see actually the tight upper bound is $O(n)$, but the operation addition is not always easy to do, so we prefer O to Θ.

3.4 Find an lower bound Big Ω on the best case run time of the following code segment. Carefully examine to see if this is a tight lower bound Big Θ

```
string bigOh2(int n)
    if(n == 0) return "a";
    string str = bigOh2(n-1);
    return str + str;
```

There are n recursion calls and each call takes $O(1)$ time. So the lower bound is $\Omega(n)$. While for tight lower bound, we have to add up the number of operations. There is a string concatenation in each recursion call, and bigOh2(n) return 2^n "a", so the tight lower bound is $\sum_{i=0}^{n} 2^i = 2^{n+1} = \Omega(2^n)$, but the operation addition is not always easy to do, so we prefer Ω to Θ.

3.5 Pual Erdos himself has a number of Zero. Anyone who wrote a paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so on. Suppose we have a database of all papers ever written along with their authors.

a. Explain how to represent this data as a graph.
b. Explain how to find the Erdos number of a given author.
c. Explain how to find all the researchers with Erdos number at most two.

Use an undirected graph because they are coauthors. Each node represents a researcher, and each edge represents a paper. Run BFS from Erdos to find all the Erdos number of all researchers, while stop BFS at level two for researchers with distance at most two.

3.6 Suppose we are interested in finding the longest simple path in a Directed Acyclic Graph. In particular, we are interested in finding a path that visits all vertices. Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

Just find the topological order of the graph. If there is a topological order, and there is a simple path follows that ordert to visit all vertices, then required simple path exists. If not, such path does not exist.

Find a longest path is not an easy thing, but if the graph happens to be acyclic, then we can use topological order to find it.