Algorithms Mini Field Guide (v1)

Written by: Krishna Parashar
Published by: The OCM
Inspired by: Raemond Bergstrom-Wood.

Basic Properties of Logarithms

$y = \log_b(x)$ iff $x = b^y$
$\log_b(xy) = \log_b(x) + \log_b(y)$
$\log_b(x) = \log_b(c)\log_c(x) = \frac{\log_c(x)}{\log_c(b)}$
$\log_b(x^n) = n\log_b(x)$ $x^a x^b = x^{(a+b)}$ $(x^a)^b = x^{(ab)}$ $x^{(\frac{1}{2})} = \sqrt{x}$
$x^{(a-b)} = \frac{x^a}{x^b}$

Basic Series

Arithmetic Series (Sequential Integers): $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Arithmetic Series (Sequential Odd Ints): $\sum_{k=1}^{n} 2k-1 = n^2$

Artihmetic Series (Square): $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$ Finite

Geometric Series: $\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}$

Infinite Geometric Series: $\sum_{k=1}^{\infty} ar^{k-1} = \frac{a}{1-r}$

Formal Limit Definition of $O, \Theta,$ and Ω

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \begin{cases} \geq 0 (\infty) & f(n) \in \Omega(g(n)) \\ < \infty (0) & f(n) \in O(g(n)) \\ = c, 0 < c < \infty & f(n) \in \Theta(g(n)) \end{cases}$$

Topological Sort $O(V+E)$

Constraints: A directed graph G is acyclic if and only if a depth-first search of G yields no back edges. Formally, we say a topological sort of a directed acyclic graph G is an ordering of the vertices of G such that for every edge (v_i, v_j) of G we have $i < j$. If DAG is cyclic then no linear ordering is possible.
Topological Sort returns a list with all nodes pointing left such that basically all parents come before any children (excluding sources). We order a graph from the **highest post number** in a decreasing order.
Thus we create singly connected component from a DAG with several strongly connected components, each a unique source and unique sink in the DAG. There are multiple topological sorting possible. Used for Runtime Compiling or Scheduling.

```
pre/post
[u[v v]u] is a Tree/Forward Edge
[v[u u]v] is a Back Edge
[v v][u u] is a Cross Edge
```

Master's Theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d) \text{ for } a > 0, b > 1, \text{ and } d \geq 0,$$

then,

$$T(n) = \begin{cases} O(n^d) & if d > log_b a \\ O(n^d log n) & if d = log_b a \\ O(n^{log_b a}) & if d < lob_b a \end{cases}$$

Branching Factor: a
Depth of Tree = $log_b n$
Width of Tree: $a^{log_b n} = n^{log_b a}$

Fast Fourier Transform $O(n \log n)$

The Fast Fourier Transform in the scope of our knowledge thus far is used for Polynomial Multiplication. We want to know the coefficient of $(p \cdot q)(x)$ knowing only the coefficients of $p(x)$ and $q(x)$. The naive way is the one we all use in Algebra which runs in $O(n^2)$. Here is a faster algorithm:

1. Take the coefficients of $p(x)$ and $q(x)$ and plug them into the FFT with the Roots of Unity (We need $2n+1$ points). $O(n \log n)$
2. From the FFT we get the roots of unity evaluated at $p(x)$ and $q(x)$ and we multiply the respected values from each root together to get $2n+1$ respective pairs of values of $(p \cdot q)(x)$. $O(n)$
3. Finally we interpolate these points using the inverse FFT to get the coefficients of $(p \cdot q)(x)$. $O(n \log n)$

Evaluation: $< values > = \text{FFT}(< coeff >, w)$
Interpolation: $< coeff > = (1/n)\ \text{FFT}(< values >, w^{-1})$

```
function FFT(A, w)
Input: Coefficient representation of a polynomial A(x)
of degree less than or equal to n1,
where n is a power of 2w, an nth root of unity
Output: Value representation A(w_0),...,A(w_n1)

if w = 1: return A(1)
express A(x) in the form A_e(x^2) + xA_o(x^2)
call FFT(A_e, w^2) to evaluate A_e at even powers of w
call FFT(A_o, w^2) to evaluate A_o at even powers of w
for j = 0 to n  1:
   compute A(w_j) = A_e(w^2j) + w^j A_o(w^2j)
return A(w_0),...,A(w_n1)
```

This process happens recursively.

Search Algorithms

Depth First Search $O(V+E)$ (Stack)

Explores all the way down to a tree, then climbs back up and explores alt paths. Use for Topological Sort and Finding Connected Components. $(pre(u) < pre(v) < post(v) < post(u))$

```
def explore(G,v): #Where G = (V,E) of a Graph
Input: G = (V,E) is a graph; v  V
Output: visited(u) is set to true for all nodes u
reachable from v
    visited(v) = true
    previsit(v)
    for each edge(v,u) in E:
        if not visited(u):
            explore(u)
    postvisit(v)

def dfs(G):
    for all v in V:
        if not visited(v):
            explore(v)
```

Previsit = count till node added to the queue
Postvsit = count till you leave the given node
A directed Graph has a cycle if and only if it a back edge found during DFS.

Strongly Connected Components $O(n)$

Us for most edge removal problems, use this to check if still strongly connected.
Properties:

- If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.
- The node that receives the highest post number in a depth-first search must lie in a source strongly connected component.
- If C and C are strongly connected components, and there is an edge from a node in C to a node in C, then the highest post number in C is bigger than the highest post number in C.

Run Time: $O(n)$
Big Picture: Topologically sort the graph; reverse the edges; topologically sort again; ff we reach a new source the resulting traversal is new SCC; continue till end of list.
Algorithm:

- Run depth-first search on G.
- Run the undirected connected components algorithm (which count uses a counter to count the number of components in G) on G, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.

Breadth First Search $O(V+E)$ (Queue)

```
Input: Graph G = (V, E), directed or undirected; vertex s  V
Output: For all vertices u reachable from s, dist(u) is set
to the distance from s to u.

def bfs(G,s):
    for all u in V:
        dist(u) = infinity
    dist(s) = 0
    Q = [s] (Queue containing just s)
    while Q is not empty:
        u = eject(u)
        for all edges (u,v) in E:
            if dist(v) = infinity:
                inject(Q,v)
                dist(v) = dist(u) + 1
```

Dijkstra's Algorithm $O(V+E)\log V$ (Binary Heap)

Objective is to find shortest path. Standard Data Structure is Priority Queue.

```
def dijkstra(G,l,s):
    for all u in V:
        dist(u) = infinity
        prev(u) = nil
    dist(s) = 0
    H = makequeue(V) # using dist values as keys
    while H is not empty:
        u = deletemin(H)
        for all edges (u,v) in E:
            if dist(v) > dist(u)+l(u,v)
```

```
        dist(v) = dist(u)+l(u,v)
        prev(v) = u
        decreasekey(H,v)
```

Bellman Ford Algorithm $O(V \cdot E)$

Objective is to find shortest path allowing for **negative edges**.

```
procedure shortest-paths(G, l, s)
Input: Directed graph G = (V, E);
edge lengths {l_e: e in E} with no negative cycles;
vertex s in V
Output: For all vertices u reachable from s,
dist(u) is set to the distance from s to u.
for all u in V:
    dist(u) = infinity
    prev(u) = nil

dist(s) = 0
repeat |V|-1 times:
    for all e in E:
        update(e)
```

Directed Acyclic Graphs

- Every DAG has a source and sink
- A directed graph has a cycle if and only if its depth-first search reveals a back edge.
- In a DAG, every edge leads to a vertex with a lower post number.
- Every directed graph is a DAG of its strongly connected components.
- If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited
- The node that receives the highest post number in a depth-first search must lie in a source strongly connected component
- If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', the the highest post number in C is bigger than the highest post number in C'.
- In any path of a DAG, the vertices appear in an increasing linearized order allowing you to run Dijkstra's Algorithm in $O(n)$

Greedy Algorithms

Definitions

A **Greedy Algorithm** always takes the cheapest least weight edge for its next step, no matter the future consequences.
A **Tree** is an acyclic, undirected, connected graph with $|V|-1$ edges (or for MST exactly $|V|-1$ edges).
A **Fringe** is all the vertices exactly one hop from you current vertex.

Kruskal's MST Algorithm $O(E \log E)$

(If Dense: $|E|$ at most $|V^2|$)
Sort edges using a sorting algorithm then repeatedly add the next lightest edge that doesn't produce a cycle (i.e. only visit new vertices).

```
Input: A connected undirected graph G = (V,E) with edge
       weights w
Output: A minimum spanning tree defined by the edges X

for all u in V:
    makeset(u)
X = {}
Sort the edges E by weight
for all edges {u,v} in E, in increasing order of weight:
    if find(u) != find(v):
        add edge {u,v} to X
        union(u,v)
```

The above algorithm utilizes disjoint sets to determine whether adding a given edge creates a cycle. Basically by checking whether or not both sets have the same root ancestor.

Properties of Trees (Undirected Acyclic Graphs)

- A tree with n nodes has n-1 edges
- Any connected undirected graph G(V,E), with $|E| = |V| - 1$ is a tree
- An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

Cut Property

Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and V-S, and let e be the lightest edge across the partition. Then $X \cup e$ is part of some Minimum Spanning Tree.

Prim's Algorithm $O(E \log E)$

Objective is to also find the MST. Alternative to Kruskal's Algorithm; Similar to Dijkstra's)
Standard Data Structure is Priority Queue. **(If Dense: $|E|$ at most $|V^2|$)**
On each iteration, the subtree defined by x grows by one edge, the lightest between a vertex in S and a vertex outside S.

```
procedure prim(G, w)
Input: A connected undirected graph G = (V, E) with weights
Output: A minimum spanning tree defined by the array prev
for all u in V :
    cost(u) = infinity
    prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0
H = makequeue (V) (priority queue with cost-values as keys)
while H is not empty:
    v = deletemin(H)
        for each {v, z} in E:
            if cost(z) > w(v, z):
                cost(z) = w(v, z)
                prev(z) = v
                decreasekey(H, z)
```

Huffman Encoding

A means to encode data using the optimal number of bits for each character given a distribution.

Set Cover Algorithm (Polynomial Time)

```
Input: A set of elements B; sets S1,...,Sm
Output: A selection of the S_i whose union is B.
Cost: Number of sets picked.

Repeat until all elements of B are covered:
    Pick the set Si with the largest number of
            uncovered elements.
```

Disjoint Sets Data Structure

Contains a function, "find" that returns the root a given set. pi refers to the parent node. $rank$ refers to the height subtree hanging form that node (number of levels below it).

- For any $x, rank(x) < rank(\pi(x))$.
- Any root node of rank k has at least 2^k nodes in its tree.
- If there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k. The maximum rank is $logn$.

```
def makeset(x): // O(1)
    pi(x) = x
    rank(x) = 0

def find(x): // O(E log V)
    while x != pi(x):
        x=pi(x)
    return x

def union(x,y): // O(E log V)
    if find(x) == find(y):
        return
    elif rank(find(x)) > rank(find(y)):
        pi(find(y)) = find(x)
    else:
        pi(find(x))=find(y)
        if rank(find(x)) == rank(find(y)):
            rank(find(y)) = rank(find(y)) + 1
```

Path Compression

```
function find(x):
    if x != pi(x): pi(x) = find(pi(x))
    return pi(x)
```

Using path compression allows for an amortized cost of O(1) for out Disjoint Set's $union(x, y)$ and $find(x)$ operations.

Union Find

Uses Disjoint Sets Data Structure
Runs in per operation $Olog * n$ which is the number of times you can take a log of n before it becomes 1 or less. It is very slow and for all practical cases is constant.
Basically if find(x) and find(y) return the same value they are in the same graph so do nothing, else add the edge. Then union(x, y).
Union: Worst case is $O(logN)$ Avg for all Ops is $O(nlog * n)$ where n is number of elements in Data Structure.