mid-term cheatsheet

Charlotte Xia

February 3, 2024

1 Divide and Conquer

1.1 Master Theorem

If $T(n) = aT(\frac{n}{b}) + O(n^d)$, where a: num of subproblems; b:subproblem size n/b; d: combining time complexity.
$O(n^d)$:combining time complexity

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log n) & \text{if } a = b^d \end{cases} \tag{1}$$

1.2 Karatsuba algorithm

For bottom layer, $3^{\log_2 n} = n^{\log_2 3} = n^{1.6}$ subproblems with linear opertion time. $T(n) = 3T(\frac{n}{2}) + O(n)$.

Algorithm 1: Karatsuba algorithm

Input: Two n-digit numbers x, y.
Output: their product
$x_L, x_R = \text{leftmost}\lceil n/2 \rceil, \text{rightmost}\lfloor n/2 \rfloor$ bits of x
$y_L, y_R = \text{leftmost}\lceil n/2 \rceil, \text{rightmost}\lfloor n/2 \rfloor$ bits of y
P1=multiply(x_L, y_L)
P2=multiply(x_R, y_R)
P3=multiply$(x_L + x_R, y_L + y_R)$
return $P1 \times 2^n + (P3 - P1 - P2) \times 2^{\frac{n}{2}} + P2$

1.3 Inversions

1.4 Selection

For **selection**, worst $O(n)$, best $O(\log n)$. For **quick sort**, if pivot chosen by median of medians, $O(n \log n)$, worst $O(n^2)$.

1.5 median of medians

$T(n) = T(n/5) + T(7n/10) + O(n)$
Assume that $T(n) \leq cn$, $T(\frac{n}{5}) < \frac{cn}{5}$,$T(\frac{7n}{10}) < \frac{7cn}{10}$, $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) < 0.9cn + bn \leq cn$. which holds when we set c to be bigger then b.

Algorithm 2: countInversions

Input: S=$(x_1, x_2, \ldots, x_n)$,A:$x_1, x_2, \ldots, x_{n/2}$; B:$x_{n/2+1}, x_{n/2+2}, , x_n$
Output: Number of Inversions: count

```
count=0
if n == 1 then
    return 0
end
count+=countInversions(A)+countInversions(B)
Maintain 2 pointers i = 1, j = n/2 + 1; C={}
while i < n/2 and j < n do
    if x_i < x_j then
        C.append(x_{i++})
    end
    else
        C.append(x_{j++})
        count+ = j − n/2
    end
end
count+=(n/2-i)×(j-n/2) # B reached end, while remaining numbers in A are bigger than all B
S=C
return count
```

Algorithm 3: Select

```
Choose an arbitrary value v among x_1, x_2, ...,
Divide x_1, x_2, ..., x_n into three subsets L, M, R
if k ≤ |L| then
    return Select(L,k)
end
else if |L| < k ≤ |L| + |M| then
    return v
end
else if |L| + |M| < k then
    return Select(R,k − |L| − |M|)
end
```

1.6 Closest Pair

If we sort by y-cordinate every merge step, $O(n \log n^2)$, can be improved to $O(n \log n)$ by pluging it into merge opertation.

Algorithm 4: FindClosestPair

Input: A list of pairs P: $p_1 = (a_1, b_1), \ldots p_i = (a_i, b_i), \ldots, p_n = (a_n, b_n)$
Output: distance of closest two pairs in P: dist

```
Sort P by x-coordinate. # O(n log n)
Divide P into two equally sized subsets A,B, the point in the middle p_mid = (a_mid, b_mid).
dist=min(FindClosestPair(A),FindClosestPair(B))
Merge A and B by y-cordinate to form C.
foreach c_i = (a_i, b_i) in C do
    if |a_i − a_mid| < dist then
        add c_i to S
    end
end
foreach s_i = (a_i, b_i) in S do
    j=i+1
    while |b_i − b_j| < dist do
        dist=min(dist, ||b_i − b_{i+1}||)
        j=j+1
    end
end
return dist
```

2 Graph

2.1 DFS

1. **Tree Edges** Edges in the DFS search: (u, v) where marked(v) = false
2. **Forward Edges** Edges that point to non-child descendants
3. **Back Edges** Edges that point to ancestors
4. **Cross Edges** All the other edges: edges point to vertices on other tree doesn't exist for undirected graph, since if (c, f) occurs it will only be tree edge.

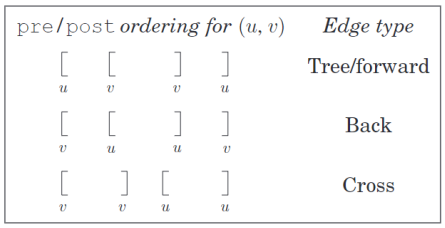

Figure 1: Directed Graph

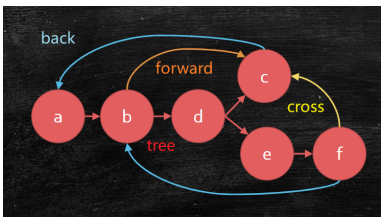

Figure 2: colored edges

Algorithm 5: DFS on Directed Graph

```
clk=0
foreach v ∈ V do
    if marked[v]=False then
        explore(v)
        clk++ # used to calculate the number of connected components
    end
end
Function explore(u):
    pre[u]=clk
    clk++
    visited[u]=false
    foreach (u, v) ∈ E do
        if marked[v]=false then
            explore (v)
        end
    end
    post[u]=clk
    clk++
```

2.2 strongly connected components

- Note that we can't simply employ DFS to find C.C., as although you find reachable points from a certain vertice v, the vertice you explore may not be reachable from each other.
- Note that unlike topological ordering, the node with the earliest post time doesn't necessarily occur in sink SCC, as the vertice with outer edges are previsited where our algorithm ends in somewhere else.i.e.,for graph **??** consider DFS in sequence 5,6,7,8, apparently, the 8 has the smallest post value, but is not in sink.
- However, the vertice with biggest post time must be in source SCC. Assume u has the largest finish time, which must be a root of one DFS tree. Suppose u isn't in source SCC, i.e. $\exists v$ s.t. (v, u) exists. v isn't in the tree of u, as (u, v) doesn't exist. ,v cannot start earlier than u, v isn't in another DFS tree.
- Consider the meta-graph, where connected components form a big node, it must be DAG, else contradict with biggest SCC, therefore, there must be a sink. Also, a way to find whether v is reachable from u is to check the connectivity of bigger SCCs, U and V.

Algorithm 6: Strongly Connected Component

Input: DAG:G
Output: num of SCC

1 Deduce G's inverse G^R from G
2 Run DFS on G^R to find source SSC, which is sink SSC of G, record it in the descending finish time as F.
3 DFS on G in the order of F.

Algorithm 7: alg:bfs

```
1  Queue Q;
2  Function bfs(,Q,u):
3      for each v ∈ V marked[v] ← false
4      Q.push(s) marked[s] ← true while Q is not empty do
5          u=Q.top();Q.pop();
6          foreach (u, v) ∈ E do
7              if marked[v]=false then
8                  marked[v]=True
9                  Q.push(v)
10             end
11         end
12     end
```

2.3 BFS

2.4 Dijkstra

Dijkstra's algorithm runs on positive-edge graph. It updates all vertice connected to the visited set and drag the closest vertex into the set. We conduct $|V|$ insert and $|E|$ update, each operation taking $O(\log n)$, $O((|E| + |V|) \log |V|)$. If there exists $x \notin SPT, v \in SPT$, s.t. $dist(s \to x \to v) < dist(v)$, $dist(x) < dist(v)$, x should be in SPT,contradiction!

Algorithm 8: Di(j)kstra

```
   Input: G=(V,E),s
   Output: dist(u) is the shortest distance from s to u
1  foreach u ∈ V do
2      dist(u)=inf
3      pre(u)=Nan # initialize
4  end
5  dist(s)=0
6  H=makequeue(V)
7  while H is not empty do
8      u=H.top();H.pop()
9      foreach (u, v) ∈ E do
10         if dist(v) > dist(u) + l(u, v) then
11             dist(v) = dist(u) + l(u, v) # update the shortest dist
12             pre(v)=u# record previous node
13         end
14     end
15 end
```

2.5 Bellman-Ford

The key idea of Bellman-Ford is that it updates all the edges (kind of 'edgewise' operation) each time (different from Dijkstra's algorithm who updates nodes one step connected to SPT), which takes $O(|V||E|)$.

Correctness of algorithm:

Algorithm 9: Bellman-Ford

```
Input: G=(V,E),s
dist[s]=0, dist[x]=inf for vertex other than s
pre(v)=Nan
while some dist[x] updates do
    # update at most |V| rounds. foreach (u, v) ∈ E do
        if dist[v] > dist[u] + w(u, v) then
            dist[v]=dist[u]+w(u,v)
            pre(v)=u
        end
    end
end
```

1. After k rounds, dist(v) is the shortest path of all k-edge-paths.($dist[u_k] \leq d(u_1, u_2, \ldots, u_k)$)
   For base step, dist[s] is the shortest dist of all 0-edge-paths.
   For induction step, suppose it is true for $k-1$ rounds, $dist[u_{k-1}] \leq d(u_1, \ldots, u_{k-1})$. By Bellman-ford update condition, $dist[u_k] \leq d(u_1, \ldots, u_{k-1}) + w(u_{k-1}, v)$, the righthand side means all the other k-edge-paths.

2. For graphs without negative cycle, all the shortest paths will have at most $|V| - 1$ edges.

   If there are more than $|V|$ paths, a vertex must be visited twice, but by erasing the whole cycle results in a shorter path, contradiction!

3. Negative cycle exists iff some dist is updated in the last($|V|^{th}$) round.

4. If no vertice updates in V^{th} round, then no vertice will be updated then. But if a single vertice is not updated.

5. If a vertex is upgraded after V rounds, then its path must include a negative cycle. A negative cycle can be found by tracing the recorded shortest path until two repeated nodes occurs. Tricks include running till the $2|V|^{th}$ round and see the vertice upgraded in $|V|^{th}$ and $2|V|^{th}$.

3 Greedy Algorithm

3.1 Prim

Adding the edge out of the previous Tree T_i with the minimum weight to form T_{i+1}.
Correctness of algorithm:

The key idea for Prim is to guarantee that the tree with n edges T_n is a subset of **global** MST T. In inductive step, if T_i is part of T, then we can construct T_{i+1}, s.t. $T_{i+1} \in T$. Consider the first edge e out of T_i, if $e = (u, v) \notin T$, then adding e to T constructs a cycle C. Since e rather than $e' \in C, e' \notin T_i, e' \in T$, $w(e') \geq w(e)$. Since T is the smallest spanning tree, $T - w(e') + w(e) \geq T$, $w(e) \geq w(e')$, therefore, $w(e) = w(e')$. Therefore, by shifting the two edges guarantees T_{i+1} is a MST component of T.
Time complexity: $O((|E| + |V|) \log(|V|))$.

3.2 Kruskal

$O(|E| \log(|E|))$ for sorting edges, and $|E|$ iterations of finding the representative element. A faster data structure is by **unified set**, which is $O(|E| \log(|V|))$

Algorithm 10: Kruskal's Algorithm

```
Sort the edge set E to ascending order.
X={}
For each u ∈ V, makeset(u). # |V| create group
For each (u, v) ∈ E in ascending order
if find(u) ≠ find(v) then
    Add edge (u, v) to X
    union (u,v)
end
Function makeset(x):
    π(x) = x
    rank(x) = 0
Function find(x):
    while π(x) ≠ x do
        π(x)=find (π(x))
    end
    return x # path Compression
Function union(x,y):
    r_x = find(x) # r_x means root(representative) of x
    r_y = find(y)
    if r_x = r_y then
        return # already in the same group
    end
    if rank(r_x) > rank(r_y) then
        π(r_y) = r_x # merge the smaller tree to the bigger tree
    end
    if rank(r_x) = rank(r_y) then
        rank(r_y) = rank(r_y) + 1
    end
```

3.3 set coverage

Denote $S_l = \{A_1, \dots A_l\}$ as the out put of greedy after l iterations, $S^* = \{O_1, \dots, O_k\}$ be **any** collection of k subsets.

$$f(S_l) \geq (1 - (1 - 1/k)^l) f(S^*)$$

For base step, $l = 1$,by greedy nature, $f(S1 = A1) \geq f(O_i)$ for all O_i. Thus $f(S_1) \geq \frac{1}{k} \sum_{O_i \in S^*} f(O_i) = (1 - (1 - \frac{1}{k})^1) f(S^*)$ This implies when calculating the sum, union elements are calculated more than once. Also, the size of the first chosen set is larger than the average size of the optimal solution. Denote $\Delta(O_i|S_t) = f(S_t \cup O_i) - f(S_t)$. $\Delta(A_{t+1}|S_t) \geq \frac{1}{k} \sum_{i=1}^{k} \Delta(O_i|S_t) \geq \frac{1}{k} \Delta(S^*|S_t)$.
For inductive step, suppose $f(S_t) \geq (1 - (1 - 1/k)^t) f(S^*)$

$f(S_{t+1}) - f(S_t) \geq \frac{1}{k}(f(S^* \cup S_t) - f(S_t)) \geq \frac{1}{k}(f(S^*) - f(S_t))$.

$f(S_{t+1}) \geq \frac{1}{k} f(S^*) + (1 - \frac{1}{k}) f(S_t) \geq \frac{1}{k} f(S^*) + (1 - \frac{1}{k})(1 - (1 - 1/k)^t) f(S^*) = ((1 - (1 - 1/k)^{t+1}) f(S^*))$.

For max-k-coverage, we at most choose k sets. $\lim_{k \to \infty}(1 - (1 - 1/k)^l) = 1 - 1/e$.
For set cover problem, it is a lnn approximation. Suppose $|S^*| = k$ is optimal(f(S^*)=n). $f(S) \geq (1 - (1 - 1/k)^{k \cdot lnn}) f(S^*) > (1 - 1/e^{lnn}) f(S^*) = n - 1$ Therefore, $f(S) = n$.

3.4 Huffman Coding

The more frequent an encoding appears, the shorter the encoding is. Denote T as the Huffman Tree, and T' the tree that switched u, v, where $len(u) < len(v)$, $w(u) > w(v)$. avg_len(T)-avg_len(T')=$w(u) \cdot len(u) + w(v) \cdot len(v) - w(v)len(u) - w(u)len(v) = (w(u) - w(v))(len(u) - len(v)) < 0$.

Next, we'll guarantee that every next step is optim, i.e. T_{i-1} is part of T. For base step,... For inductive step: If we merge A and B in T. If A is paired with C in T^*, then $w(C) > w(A)$,$w(B) \leq w(C)$. If $w(B) = w(C)$, changing B and C preserves avg_len. If $w(B) < w(C)$, $len(B) \geq len(C)$ by above proof. If $len(B) == len(C)$, swapping B and C also preserves avg_len. If $len(B) > len(C)$, D is not processed, $w(D) > max(w(A), w(B))$, but $len(D) = len(B) > len(C)$, contradict with the fact that T^* is optimal(by above proof).

Another intepretation is by sorting inequality, namely, the sum of $\sum ordered\_sequence \leq \sum random\_sequence \leq \sum inversed\_sequence$.

time complexity: By using binary-heap, every time we conduct pop twice to obtain the top two min vertice and a push to add the new parent vertice in the tree, which takes $O(n \log n)$.

3.5 k-center

- **Algorithm:** Iteratively pick the center farthest to the existing centers. Pick the first by random. by Dijkstra for weighted graph and BFS for unweighted.

- The algorithm is **2-approximation**.

  Denote $OPT = \max \min d(o_1, v)$. $A = \{a_1, \dots a_k\}$ for algorithm result, $X_i = \{v_1, \dots v_i\}$ as the set of vertice closest to o_i, optim solution $O = \{o_1, \dots, o_k\}$

  Case 1:Suppose $A \cap X_i \neq \varnothing$ for all i. Then each X_i contains a center, as the biggest length between two vertice in X_i is smaller than $2OPT$ by triangle inequality.

  Case 2:Suppose $\exists A \cap X_i = \varnothing$. Suppose we choose a_j and $a_r \in X_i$. Let a_r be the second center chosen in X_i. By greedy choice, before a_r was chosen, the algorithm answer $ALG = d(A, a_r) \leq d(a_j, a_r)$. Additionally, $d(a_j, a_r) \leq d(a_j, o_i) + d(a_r, o_i) \leq 2OPT$.

3.6 max-cut

- Start with any partition $\{A, B\}$. If moving a vertex u from A to B or from B to A increases $c(A, B)$, move it. Terminate until no such movement is possible

- **Time Complexity** checking whether to update takes $O(|E|)$, every update you earns at least an edge, so it takes $O(|E|)$ rounds. The total algorithm is $O(|E|^2)$, polynomial!

- **Approximation** When the algorithm terminates, $\forall u$, at least 0.5deg(u) edges are in the cut, which leads to a 0.5-approximation.

$$c(A, B) \geq \frac{1}{2} \sum_{u \in V} \frac{1}{2} deg(u) = \frac{1}{2}|E|$$