Chapter 3 | Graphs

- Adjacency List: $O(|V| + |E|)$
- Adjacency Matrix: $O(|V|^2)$

```
function runner_dfs(G, v):
    for all v in V:
        visited(v) = false
    for all v in F:
        if not visited(v): dfs(v)

function dfs(v):
    visited(v) = true
    previsit(v) # used for labeling
    for each child:
        if not visited(child): dfs(child)
    postivisit(v) # used for labeling
```

Pre/Post Ordering Edge types

$$[\ \text{pre(u)}\ [\ \text{pre(v)}\ ,\ \text{post(v)}\ ]\ \text{post(u)}\ ] - \textbf{Forward}$$
$$[\ \text{pre(v)}\ [\ \text{pre(u)}\ ,\ \text{post(u)}\ ]\ \text{post(v)}\ ] - \textbf{Back}$$
$$[\ \text{pre(v)}\ ,\ \text{post(v)}\ ], [\ \text{pre(u)}\ ,\ \text{post(u)}\ ] - \textbf{Cross}$$

Graph Algorithms

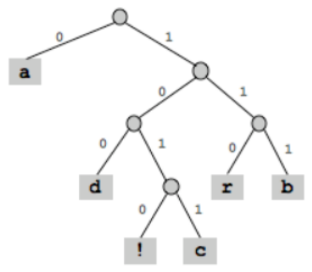
- Topological Sort - $O(|V| + |E|)$ - DFS with post ordering. Reverse post ordering to get a topological sort.
- Kosaraju's / SCC Algorithm - $O(|V| + |E|)$ - Run **DFS** search on G^R w/ post ordering. Run **BFS** on highest post order on G, everything it reaches is an SCC. Repeat on node w/ next highest post roder.
- Dijkstra's Algorithm - $O(|V|^2)$ - Minimum distance between a node and all other nodes reachable from it. Add all verticces to an array / heap with values of infinity. Start at a source node. For all the edges of a node, add the current distance (the value for the current vertex) to all the edge weights and update nodes within the heap to this new value. Pop from heap. Repeat this process.
- Bellman-Ford - $O(|V|^2)$ - Minimum distabce between a node and all other nodes reachable from it. Works on negative edge weights, which Dijkstra's does not.
- Kruskal's Algorithm - $O(E \log V)$ - Finds MST for an undirected, weighted graph - Sort (low $\rightarrow$ high) edges by weight. Add edges s.t. cycles are not formed.
- Prim's Algorithm - $O(E \log V)$ - Finds MST for an undirected, weighted graph - Breadth first search except have a heap sorted on edge weights. Pop from the top of heap. Add edge if node has not been visited.

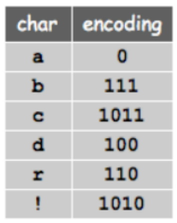
Graphs Properties

- For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other.
- Reverse post ordering will give a linearized graph. Called a topological sort. $O(|V| + |E|)$
- 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 DAG has at least one source and at least one sink
- Every directed graph is a dag of its strongly connected components.
- If the **dfs** function is started at a 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'.

Huffman Encoding

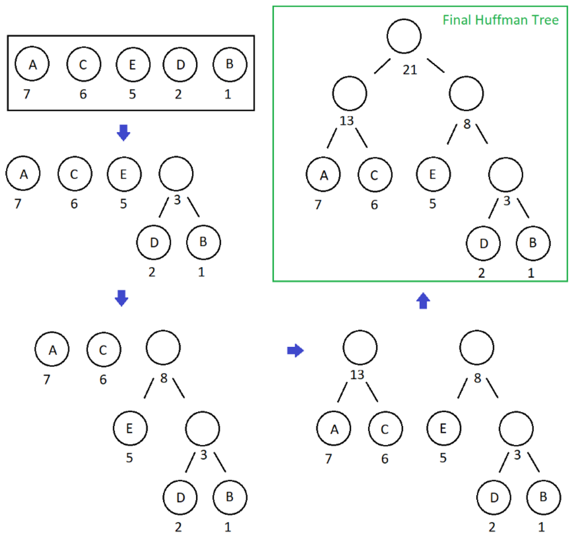
A binary tree where more frequent characters have smaller encodings.





char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

1. Create key, value pairs for the characters and their frequencies. Sort low to high based on frequency.
2. From the collection, pick out the two nodes who has the smallest sum of frequency.
3. The root has a sum of the frequencies. Add to the graph.
4. Repeat this process until all probabilities are used.



Dynamic Programming

Procedure for finding and writing a DP solution.

1. Subproblem: Define what a index of the array represents.
2. Recurrence: Define how to find a solution of a subproblem of your subproblems based on smaller subproblems.
3. Algorithm: **1.)** Define size and default values of your array. **2.)** Set base case values. **3.)** Apply recurrence ot the array. **4.)** Give the return value. May be a single index, or may involve iterating through the array.
4. Correctness: Doesn't and shouldnt be a formal proof. Give a description of the cases for the recurrence, and why they model the problem.
5. Running time: Doesn't need to be long or formal, give a brief description of why/what the running time is.

Knapsack Variations

– Total capacity `W`. Also w_i, v_i is weight and value for object i.
– Knapsack w/ Repetition: If $\exists$ a solution for $K(W)$ then removing item i gives the solution for $K(W - w_i)$. Thus:

$$K(w) = \max_{i:w_i \le w} \{K(w - w_i) + v_i\}$$

– Knapsack w/out Repetition: $K(w, j)$ is the maximum value achievable using a knapsack of capacity w and items $1, ..., j$.

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Linear Programming

A broad set of problems that take in constraints and optimization criterion as linear functions. This boils down to assigning real values to a set of variables such that they satisfy a set of linear equations/inequalities and maximize/minimize a linear objective function.

– **Duality**: Every linear maximization problem has a **dual** minimization porblem.
– **Duality Theorem**: If a linear program has a bounded optimum, then so does its

dual, and the two optimum values coincide.

– **Max-flow min-cut theorem**: size of maximum flow $==$ capacity of smallest min-cut.
– **Bipartite matching**: Given a bipartite graph, find a set of edges st everything is paired exactly once. Add a source to one side and a sink to other and all edges have a weight of one. $/exists$ a perfect matching iff network flow $==$ number of couples

Primal Linear Program:

$$\begin{aligned} \min/\max\{\alpha x_1 + \beta x_2 + \gamma x_3\} \\ k_{1,1}x_1 + k_{1,2}x_2 + k_{1,3}x_3 \le a \\ k_{2,1}x_1 + k_{2,2}x_2 + k_{2,3}x_3 \le b \\ k_{3,1}x_1 + k_{3,2}x_2 + k_{3,3}x_3 \le c \\ x_1, x_2, x_3 \ge C \end{aligned}$$

The Dual:

$$\begin{aligned} \max/\min\{ay_1 + by_2 + cy_3\} \\ k_{1,1}y_1 + k_{2,1}y_2 + k_{3,1}y_3 \ge \alpha \\ k_{1,2}y_1 + k_{2,2}y_2 + k_{3,2}y_3 \ge \beta \\ k_{1,3}y_1 + k_{2,3}y_2 + k_{3,3}y_3 \ge \gamma \\ y_1, y_2, y_3 \ge C \end{aligned}$$

NP-Complete

– **NP** - all search problems, nondeterministic polynomial time
– **P** - can be solved in polynomial time
– **NP-Complete** - subset of problems in NP but not in P
– Changing a search problem into an optimization problem is quite easy as the two reduce into one another.
– **SAT** - Given a Boolean formula in conjunctive normal form (CNF), set each boolean variables to a True/False value such that all the clauses are satisfied. If each clause has at most three boolean values then it is **3SAT**.

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

– Euler / Rudrata - Can every vertex within a graph be reached without visiting a vertex twice?
– **Traveling Salesman** - Given a graph G and a maximum cost M, find a path through n vertices such that the total distance traveled is less than M.
– Longest Path - Finding the path with maximum length between two vertices.
– 3D matching - Given three disjoint sets and a set of triplets representing edges (a, b, c) where $a \in A$, $b \in B$, and $c \in C$. Find a subset of the triplets such that each point in $A \cup B \cup C$ appears exactly once in the triplets.
– Knapsack - Given a set of items with weights and values, find which items to fill a knapsack such that the knapsack does not exceed a certain weight.
– Independent Set - Given a graph G with a set of edges E, determine what is the largest number of vertices such that there does not exist an edge between any two vertices.
– **Integer Linear Programming** - Linear proramming except the answer must be integer values. This turns linear programming into a NP-complete problem.