9.0

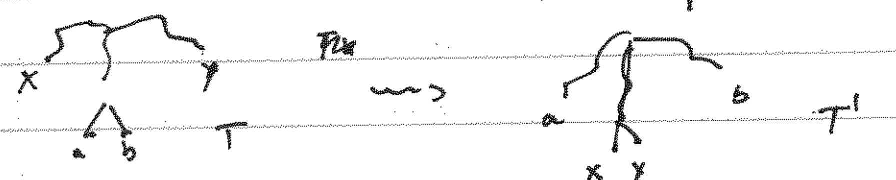
Last week we gave some examples of greedy algorithms.
We still have to prove two lemmas concerning Huffman's alg.
Then we will start discussing graphs and graph algorithms.

9.1

~~8.8~~

Lemma: Let x, y be with lowest freq. Then, there exists an optimal prefix-code C in which codewords for x and y have the same length and they differ in only ~~one bit~~ the last bit

Pf: We will start with any optimal tree and get another optimal tree in which this property holds.
Let T be an optimal tree and let a, b be siblings at maximal depth
Assume w.l.o.g. $x.freq \le y.freq$, $a.freq \le b.freq$.
We would like to replace x with a and y with b.



what will be the diff between the encodings:

$$\sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) =$$

$$x.freq\,(d_T(x) - d_{T'}(x)) + a.freq\,(d_T(a) - d_{T'}(a)) + y.freq\,(d_T(y) - d_{T'}(y)) + b.freq\,(d_T(b)$$

Notice $d_T(x) = d_{T'}(a)$ $d_{T'}(x) = d_T(a)$ same for y, b. $\Longrightarrow$

$$= (a.freq - x.freq)(d_T(a) - d_{T'}(a)) + (b.freq - y.freq)(d_T(b) - d_{T'}(b))$$

$$= \underbrace{(a.freq - x.freq)}\ \underbrace{(d_T(a) - d_T(x))}_{\ge 0} + (b.freq - y.freq)\underbrace{(d_T(b) - d_T(y))}_{0 \le} \ge 0$$

By assumption $x.freq \le y.freq$ are smallest. ~~So if $(a,b) \ne (x,y)$ then at least one of them is different~~ Thus, the entire sum is non negative.
Thus T' at least as good as T.
The proof also gives that x, y are siblings at maximal depth! $\blacksquare$

9.2

~~8.1~~

We are ~~almost done~~. We saw that in ~~the~~ some optimal code, x,y will be siblings at max depth. But ~~is the~~ the question is, do we get an optimal code by our process?

Lemma: Let C' be the new alphabet after "merging" x,y. Let T' be any optimal ~~post~~ prefix-code / Tree for C'. Let T be obtained by adding two ~~leaves for~~ children to z in the obvious way. Then T is optimal for C.

Proof: ~~The~~ Let $B(T'), B(T)$ be the costs of T', T.

Then $B(T) = B(T') + x.freq + y.freq$ (everything stayed the same except z got two new children)

$\Rightarrow \quad B(T') = B(T) - x.freq - y.freq.$

Assume for a contradiction that T is not optimal. Let S be a better tree. wlog S has x,y as siblings. ~~at~~ Let S' be same as S with z instead of x,y.

$$B(S') = B(S) - x.freq - y.freq \underset{\substack{\uparrow \\ \text{our assumption}}}{<} B(T) - x.freq - y.freq = B(T')$$

contradicting T''s optimality.

The two lemmas imply that Huffman is optimal.

9.3

We now move to graphs and related algorithm.

A graph has a set of vertices V (in our case V will always be finite) and a set of edges $E \subseteq V^2$. If $(u,v) \in E$ we imagine an edge going from u to v.

~~We say the graph~~ A graph as we described is call a directed graph.

Notice that we can have both $(u,v) \in E$ and $(v,u) \in E$.

We can also have a self loop - $(u,u) \in E$.

A graph is undirected if edges are unordered pairs $\{v,u\}$.

The degree of a vertex is the number of neighbors in an undir. graph.

In a directed graph we have indegree and out-degree

A path is a sequence $v_1, v_2, \ldots, v_k$ where $\forall i$ $(v_i, v_{i+1}) \in E$.

v is reachable from u if there is a path from u to v.

A path is simple if all vertices are distinct.

A cycle is a ~~non~~ path from u to u ~~with~~

~~A graph is a~~ An undirected graph is connected if there is a path between any two vertices.

A dir. graph is strongly connected if there is a dir. path between any two vertices. (both directions).

The connected component of u in an undir. graph is all vertices reachable from u.

The strongly connected component of u in a dir. graph is the set of all vertices that have a dir. path to and from u.

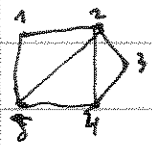
9.4

We would like to work with graph so we have to find a way to represent them that is efficient and easy to work with.

One representation is by the adjacency matrix and the other by adjacency list.

Adj-list: for every u we have a list of all v ~~that~~ s.t. $(u,v) \in E$

E.g. for an undir graph.



1 → 2 → 5
2 → 1 → 4 → 5 → 3
3 → 2 → 4
4 → 2 → 5
5 → 4 → 1 → 2

The adjacency matrix is a $|V| \times |V|$ matrix A with $A_{ij} = 1 \Leftrightarrow (i,j) \in E$

$$\begin{array}{c c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}
\end{array}$$

Adj-list is shorter but "harder" to find whether $(u,v) \in E$. (have to go through u's list). They are of size $|E|$ and $|V|^2$ so if $|E| \approx \Omega(|V|^2)$ they are of similar sizes.

We now present two canonical alg for searching graphs, which are at the core of many other algorithms.

9.5

Breadth First search (BFS)

BFS is an alg. that given a graph $G=(V,E)$ and a source vertex $s \in V$ explores the graph and discovers every vertex reachable from s along with its distance from s.
The distance of u from s is the length of the shortest path from s to u.
The alg. also produces the BFS tree with root s and all reachable vertices.
It works both for directed and undirected graphs.

The alg. works by first discovering s's neighbors, then their neighbors etc. At each step the frontier advances ~~by~~ between discovered and undiscovered vertices across the breadth of the frontier.

Give a picture.

We shall color vertices white if they were not discovered yet, ~~the~~ gray if they are at the frontier and black if we are done with them - i.e. we exposed all their nbrs.

The alg. will use a queue - a first-in first-out data structure.
Enqueue puts an element at the end of the queue and Dequeue extracts the ~~top~~ first element from the queue.

9.6

```
BFS(G,s)
For each u ∈ G.V \ {s}
    u.color = white
    u.d = ∞                          distance
    u.π = NIL                        predecessor
s.color = gray
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q,s)
While Q ≠ ∅
    u = Dequeue(Q)
    For each v ∈ G.Adj[u]
        If v.color == white
            v.color = gray
            v.d = u.d+1
            v.π = u
            Enqueue(Q,v)
    u.color = black
```

Run time: Each vertex enters Q at most once (color change). When we dequeue a vertex we go over all its neighbors. Thus run time is $O(V+E)$

<u>Claim</u>: BFS discovers every vertex v that is reachable from s, upon termination $v.d = dist(s,v)$, and a shortest path from s to v is given by shortest path from s to $v.\pi$ and the edge $(v.\pi, v)$.

9.7

Furthermore, let

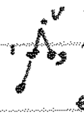
$$V_\pi = \{ v \in V \ : \ v.\pi \neq NIL \} \cup \{s\}$$
$$E_\pi = \{ (v.\pi, v) \ : \ v \in V_\pi \setminus \{s\} \}$$

This is the predecessor subgraph of G and s.

__Claim:__ The predecessor graph produced by BFS is a tree consisting of all reachable vertices from s, and for all $v \in V$ it contains a unique simple path of length $v.d$ from s to v.

Both claims are not hard to prove by induction.

Depth-First-Search (DFS)

This is a different searching alg. that works by going as deep as possible before exploring other vertices.
Basically, DFS explores edges out of the most recently discovered v (so e.g. v's first grandchild will be discovered before v's 2nd child: ). ~~The~~ Once all of v's edges have been explored we backtrack to explore edges leaving the vertex from which v was discovered.

For each $v \in V$ we will also record the time when it was discovered - $v.d$ and the time when we finished with it - $v.f$.

9.8

```
DFS(G)
for each u ∈ G.V
    u.color = white
    u.π = NIL
time = 0
for each u ∈ G.V                    If we just want the DFS trees
    if u.color == white              we should run DFS-visit(G,s)
        DFS-visit(G,u)

DFS-visit(G,u)
time = time+1
u.d = time                          u is discovered
u.color = gray
for each v ∈ G.Adj[u]
    if v.color == white              we explore along the edge (u,v)
        v.π = u
        DFS-visit(G,v)
u.color = black                     we finished exploring all u's nbrs.
time = time+1
u.f = time
```

Run-time of DFS is also $\Theta(V+E)$.

~~Indeed, time is updated whenever~~

Indeed we run DFS-visit only when we reach a white neighbor. Thus, it is called once for each vertex. (a vertex is painted gray immediately at DFS-visit).

Then in DFS-visit the total number of ~~executions~~ commands to execute is $|Adj[v]| + O(1)$

As each DFS-visit(G,v) is run only once the total # of operations is $O(\sum |Adj[v]|) = O(E+V)$

Some important properties of DFS:

1. (V, E_π) is a forest
2. u is ancestor of v in the DFS forest iff at time u.d there is a u→v path consisting of only white vertices (before changing u's color to gray)
3. If u is ancestor of v then $[v.d, v.f] \subseteq [u.d, u.f]$. If they are unrelated the interval are disjoint. In fact, this is iff statement.