DAGs and Topological Sort, Huffman Codes

Recap: DAGs

- Directed Acyclic Graphs
- Like trees, except that nodes can have “multiple parents”
- Edges are all directed “down” with no edges going “up”
- Commonly used to model real data, for example nodes are “tasks” and edges mean one task must be completed before another

Remember Terminology

- `Degree` : in an undirected graph, the number of edges connected to a node
- `Indegree` : in a digraph the number of edges that point into this node
- `Outdegree`: in a digraph number of edges pointing out from this node

Topological Sort

- Given a project with tasks represented as a DAG, in which order can I complete the tasks?
- The order might not be unique!

Topological sort algorithm

```
Q = empty queue
for each node:
    if indegree of node == 0: enqueue it

while !Q.empty:
    n = dequeue(Q)
    output n
    for each neighbor of n:
        neighbor.indegree -= 1
        if neighbor.indegree == 0: enqueue it
```

More terminology

- Given a graph, a “subgraph” is a subset of nodes and edges from the graph
- The subset of edges can only connect edges between nodes in the subset of nodes
- Basically, we take a graph and throw away some vertices and all edges connected to them, and throw away any other edges we like

Minimum Spanning Tree

- Given a connected graph, the “minimum spanning tree” is a subgraph that is connected, with tree topology
- It is the minimum set of edges needed for the graph to remain connected
- We usually care about the MST of weighted graphs and want to minimize the sum of weights of all edges in the tree

Prim's Algorithm

```
Start with any vertex
while there are vertices not yet in the tree:
    pick the lowest cost edge from one of those vertices to any node currently in the tree
```

Kruskal's Algorithm

```
Create a "disjoint set" for each node
for each edge(u,v), sorted by weight
    if set(u) ≠ set(v):
        merge set(u, v)
        add the edge to the tree
```

Greedy Algorithms

- These are both “Greedy algorithms” which make the “locally optimal” choice at each step
- Prim's algorithm and Kruskal's algorithm happen to produce the globally optimal spanning trees
- Many greedy algorithms do not produce globally optimal results because a sequence of a couple seemingly suboptimal steps produces a better overall result

Huffman Codes

File Compression

- There's many ways of representing the same information, for example .jpg vs .png images
- These different representations may require less data, which reduces the cost for storing or sending the data
- We'll look at the particular case of text data (though the ideas apply more broadly)

ASCII Encoding

- For latin text (ie, all the letters/characters on a North American keyboard), text is stored encoded as ASCII
- Each character is stored in a byte, so ASCII can represent 256 (sort of, more like 127) characters
- If I'm sending an email, I will only be using a small subset of the available ASCII characters, so I might not need to use all 8 bits of a byte for each character

Compression

- Compression is the process of coming up with a different encoding that stores the same data with fewer bits
- For this particular application, we're going to try to encode (some) characters in fewer than 8 bits
- If we can represent the most common characters in 1, 2, or 3 bits, then even if some rare characters take more than 8 bits in our encoding, we end up saving space
- We're going to design an algorithm to efficiently come up with “good” encoding system that tries to minimize the length of the encoded representation

Input: Character weights

- We'll start by counting the occurrences of each character in the message we're encoding (which data structures would you use to do this?)
- We want to come up with an encoding that uses fewer bits for more frequent characters
- Unfortunately, this will require us to use longer encoding for less frequent characters (maybe worse than the original 8-bit encoding!)

Prefix Free Codes

- We need our encoding to be unambigous and efficient to decode and convert back to ASCII
- One way to do this is to make sure that no character encoding is a **prefix** of another encoding
- As an example: if e was encoded as `0` and a was encoded as `01`, What does the encoded string `0001010` mean? What if `t` is represnted by `10`?
- We want a “prefix-free” encoding so that we can easily tell the “end” of each encoded character
- We'll solve this problem by creating a tree where paths in the tree specify encodings

Binary Trie (pronounced “try”)

- A binary trie is a binary tree where the leaf nodes store the ASCII characters we want to encode
- The path from the root to a leaf specifies its encoding: every time we go “left” we append a `0` to the encoding, and every time we go “right” we append a 1
- What about this structure guarantees our codes are prefix-free?

Encoding

- Once we've built the tree, we can compute and store the encoding for each character (what would be a good data structure for this?)
- We produced the encoded message by looking up the encoding for each character, and concatenating them all together
- This is actually a bit tricky in code because dealing with data smaller than 1-byte in size is a pain, but it's doable and fast

How to build the tree?

- We'd like high-weight nodes near the top, and low weight nodes near the bottom
- What should the weights of internal nodes be?
- We know what should be in the leaves, so bulding “bottom up” seems like a promising strategy
- Like with heapify, we'll start with small valid tries, and then repeatedly merge them until we have a single trie with all the characters we want to encode

Tree Construction

```
pq = priority queue keyed on weight, ties broken by ascii value of the leftmost node
for each character + count:
    make a node with the given character + weight = count, add it to the priority queue
while pq.size > 1: //more trees to merge
    t1 = pq.extractMin(), t2 = pq.extractMin()
    n = new node(t1.weight + t2.weight, t1, t2)
    pq.add(n)
final tree = pq.extractMin() //1 tree with all nodes in it
```

Does it compress?

- Does this actually reduce memory?
- Are the encodings for each character actually shorter than their ASCII representation?
- A shorter encoding for one character requires longer encodings for another. Imagine one character has a 1-bit encoding (let's it's the left child of the root), half of the potential “leaf spots” in the tree are eliminated since there can't be any othe leaves in the left half of the tree!
- Depending on weights, we only need to have 8 different characters in our dictionary between some of them have an even longer encoding than their original ASCII!

Overhead

- We also need to include the character occurrences in our compressed file so that the decompressor knows how to build the huffman tree
- In a basic implementation, we'd need 5 bytes (1 ascii character and a 4 byte int) for each unique character
- For long messages, this overhead is small relative to the message itself

When is it worth it?

- Huffman coding works best when characters are unequally distributed in the message
- English text is a good example: vowels, t, s, d, etc appear much more often than q or z, so even if q has a 12 bit encoding, we'll still achieve good compression

Code Demo