CSC2710 Analysis of Algorithms — Fall 2022
Unit 12 - Greedy Algorithms

Book: §16.1 – 16.3; §23.1 – 23.2; §24.3

12.1 Introduction to Greedy Algorithms

- Sometimes using dynamic programming to solve a problem is overkill to solve an optimization problem.
- One alternative is using *greedy algorithms* to solve an optimization problem.
  - A greedy algorithm always makes the choice that looks best at the current step.
    - This is often referred to as the *greedy choice*
- **Important:** Greedy algorithms do not yield optimal solutions for all problems.
- There are several key facts about greedy choices: They must
  - be feasible. In other words, they must satisfy the problems constraints.
  - be locally optimal. In other words, the best local choice among all feasible choices available on a given step.
  - be irrevocable. In other words, once made, the choice cannot be changed on subsequent steps of the algorithm.
- We can design greedy algorithms using three steps.
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is *always* safe.
  3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.
     - This is sometimes referred to as exhibiting the *greedy choice property*.
     - **Greedy Choice Property**: we can assemble a globally optimal solution by making locally optimal (greedy) choices.
- To illustrate the greedy algorithm design technique we consider the problem of making change

  Definition 1 (Change Making Problem)**.**
  Input*: A specific amount of change n owed to a customer and a sequence of m different coin denominations $d_1 > d_2 > \cdots > d_m$.*
  Output*: The fewest amount of coins such that $\sum_{i=1}^{m} c_i d_i = n$ where $c_i \geq 0$ denotes the number of coins of denomination i.*
- How would you make change for $n = 45$ cents where $d_1 = 25$(quarter), $d_2 = 10$(dime), $d_3 = 5$(nickle), and $d_4 = 1$(penny)?
- **Practice**: How would you solve this in general?
- **Practice**: What is the greedy choice for change making?
- This algorithm is greedy and optimal. We are guaranteed to always get the smallest number of coins.
  - While the correctness is easy to see, it is harder to prove that the problem exhibits the greedy choice property.
  - Generally we prove this through the use of induction.

12.2 Prim's Algorithm

- We return to your old friend the minimal spanning tree.
- We will review some definitions to help.
  - **Spanning Tree**: the spanning tree of an connected undirected graph $G = (V, E)$ is an acyclic subgraph that contains all the vertices of the graph.
    - An acyclic graph is the technical term for a tree.
  - **Minimum Spanning Tree**(MST): the spanning tree of a weighted connected undirected graph $G = (V, E)$ such that the sum of the weights of all spanning tree edges are minimal.
- We formally define the MST problem as follows:

  Definition 2 (Minimal Spanning Tree (MST) Problem)**.**
  Input*: An undirected weighted graph $G = (V, E)$ where the edge weights are given by a function $w : E \to \mathbb{R}$.*
  Output*: An acyclic subset $T \subseteq E$ that connects all vertices in V such that the weight of the graph T,*

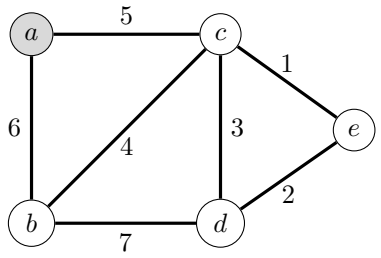
  $$\sum_{(u,v)\in T} w(u, v)$$

  is minimized.

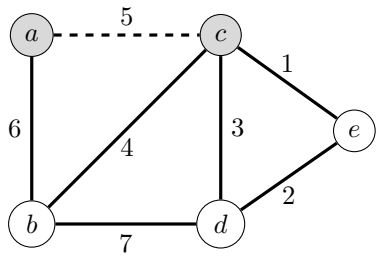
- Applications of MSTs include
  - Routing in computer networks
  - Data point cluster detection
  - Classification purposes in a wide range of scientific disciplines
  - For approximations to more difficult network problems (e.g., Traveling Salesman problem)
- To solve the MST problem using Prim's algorithm[1] we "grow" a spanning tree.
- We maintain this notion of a *frontier* which are essentially the current leaves of the tree and successively add edges to the the frontier thus decreasing the number of unconnected vertices.
- Overview
  1. Select a vertex $s \in V$ as the starting point.
  2. Repeat until all edges are added to the tree $T = (V_T, E_T)$.
     (a) Add a *light edge*, (u, v) where $u \in V_T$ and $v \in (V \setminus V_T)$. Specifically,
        i. $V_T = V_T \cup \{v\}$.
        ii. $E_T = E_T \cup \{(u, v)\}$.
- What is a *light edge*? This requires a few definitions
  - **Graph Cut**: A cut of an undirected graph $G = (V, E)$ is partition of V into two sets S and $V \setminus S$. We will often denote a cut as $(S, V \setminus S)$.
  - **Cross Edge**: An edge $(u, v) \in E$ that crosses the cut $(S, V \setminus S)$. In other words, one endpoint of the edge is in S and the other endpoint is in $V \setminus S$.
  - **Light Edge**: A cross edge $(u, v) \in E$ that is the lightest, smallest weight, of any edge that crosses the cut.

[1] Sometimes called the Prim-Jarník algorithm as Prim rediscovered (1957) Jarník's original algorithm (1930).

- As shown above our cut is $(V, V \setminus V_T)$.
- We must also add a bit of book keeping to our overview algorithm. Every vertex $v \in (V \setminus V_T)$ must contain information about the lightest edge connecting v to a vertex $u \in V_T$. Specifically,
  - The label for vertex u (known as the predecessor)
  - The distance (weight) of edge (v, u), denoted $w(u, v)$.
- Now to detect the light edge it is a simple as maintaining a minimum priority queue of nodes in $V \setminus V_T$ that is keyed on the distance (edge weights).
  - A minimum priority queue can be implement in a very similar manner to a max priority queue – simply change the relational operators appropriately.
  - Removing a node u from the priority queue in effect moves it to set V_T thus affecting the graphs cut.
  - For all vertices $v \in (V \setminus V_T)$, we update the priority queue by determining if $w(u, v)$ is lighter than the current lightest edge connecting v to the tree.
    - This technically achieved by changing the priority of an element in the priority queue, an $O(\lg n)$ operation.
    - We also assign the appropriate predecessor.
- Notice our greedy choice in this algorithm is always selecting the light edge every time we add a vertex to the tree.
- Let's look at example of Prim's algorithm on the graph given below with starting vertex $s = a$ (shaded edges denote vertices in V_T):



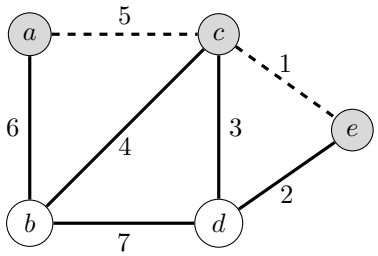
1. We start with a queue of $Q = \langle (c, 5, a), (b, 6, a), (d, \infty, \perp), (e, \infty, \perp) \rangle$
   - Each tuple, in Q, consists of the vertex, current minimum weight to the a tree vertex, and the tree vertex it shares its minimum weight edge with.
   - $\perp$ denotes the absence of a predecessor
   - ∞ denotes the absence of an edge connected the vertex to the tree.
2. We select the minimum element from the queue Q and get $(c, 5, a)$. Our graph becomes,



The queue becomes $Q = \langle (e, 1, c), (d, 3, c), (b, 4, c) \rangle$.

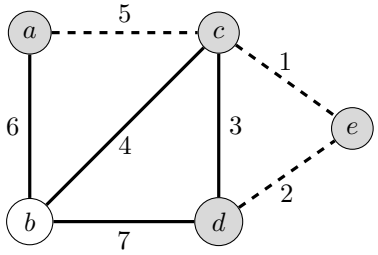
- Dashed edges in the graph represent the tree edges

3. We select the minimum element from queue Q and get $(e, 1, c)$. Our graph becomes,



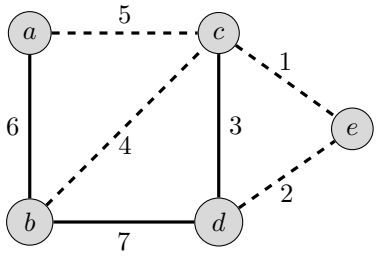
Updating the queue we get $Q = \langle (d, 2, e), (b, 4, c) \rangle$

4. We select the minimum element from queue Q and get $(d, 2, e)$. Our graph becomes,



Updating the queue we get $Q = \langle (b, 4, c) \rangle$

5. We select the minimum element from queue Q and get $(b, 4, c)$. Our graph becomes,



Since the Q is now empty, there are no more vertices in $V \setminus V_T$ we have a complete spanning tree.

- If the graph is represented as an adjacency list and the priority queue is implemented as a min-heap the running of Prim's algorithm consists of
  - $|V| - 1$ removals of the minimum element in the heap.
  - $|E|$ updates of edges in queue (each edge is only looked at, at most, once)
  - Each operation runs in time $O(\lg n)$ where n is the size of the queue (i.e., $|V|$)
  - Therefore, we have $(|V| - 1 + |E|) O(\lg |V|) = O(|E| \lg |V|)$
    - Note, that there is a fact from graph theory that states if a graph is connected $|V| - 1 \leq |E|$.
- Let's consider the following theorem

Theorem 1. *Prim's algorithm always yields the minimal spanning tree.*

- The proof will give us a flavor of how greedy choice algorithms are shown to be optimal.

  Proof. By way of induction on the number of subtrees $i = 0, 1, \ldots, n-1$ generated by Prim's algorithm, we will show that each subtree is part of the same minimal spanning tree T. We denote by T_i the subtree consisting of $i + 1$ vertices.

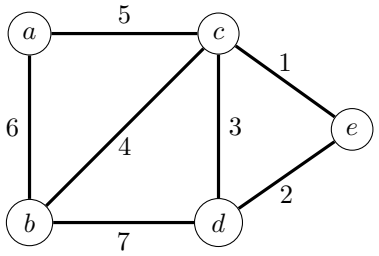
  Base Case: Let T_0 denote the tree consisting of only one vertex. Trivially, this part of *any* minimal spanning tree. Therefore it is certainly part of T

  Inductive Step: We assume the inductive hypothesis which states that T_{i-1} is part of T. We show that T_i, generated from T_{i-1} by Prim's algorithm, is also part of T. We assume, by way of contradiction, that T_i can not be part of T. Let (u, v) be the light edge connecting vertex u in T_{i-1} with vertex v *not in* T_{i-1} that will expand T_{i-1} to T_i (cf. Prims algorithm). By assumption, (u, v) can not belong to any minimal spanning tree, including T. Therefore, if we add (u, v) to T we must form a cycle (break the tree property). Since we have a cycle, it must be the case there exists an edge (u', v') connecting a vertex u in T_{i-1} to a vertex v not in T_{i-1}. We know this because T_{i-1} is part of MST T by assumption. If we delete (u', v') from the cycle, we obtain another minimal spanning tree of the entire graph whose weight is less than or equal to the weight of T since $w(u, v) \leq w(u', v')$. Hence, the tree is a MST which contradicts the assumption that no MST contains T_i.

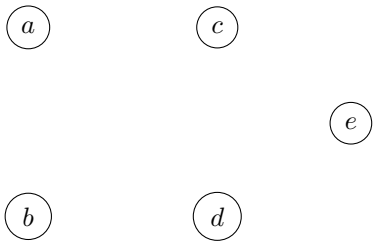
  □

12.3 Kruskal's Algorithm

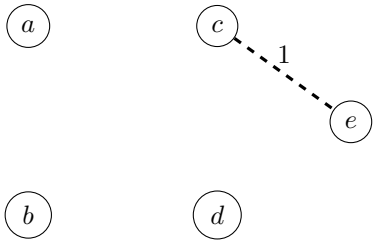
- Kruskal's Algorithm is another algorithm for finding a MST.
- Discovered by Joseph Kruskal in 1956 (when he was a second year graduate student)[2].
- The basic idea is to take the graph $G = (V, E)$ and initially treat V as a set of distinct trees, thus forming a *forest* G'. We proceed as follows:
  1. Order the edges E in increasing order by weight for a sequence $S = \langle s_1, s_2, \ldots, s_n \rangle$.
  2. Repeat until we have one tree.
     (a) Get the minimum edge (u, v) from S.
     (b) If u, v are not part of the same tree in the forest G', connect the two trees with edge (u, v)
- Alternatively, we don't have to deal with forest and simply just add edges of S, in order, to G' provided that before adding the edge to G' we check to see that the addition will not create a cycle in G'.
  - Any edge that would create a cycle is simple skipped.
- Let's look at a Forest based example of Kruskal's algorithm on the graph given below:



1. We start with the sequence of sorted edges
$$S = \langle (c, e), (d, e), (c, d), (c, b), (a, c), (a, b), (b, d) \rangle$$
and the forest shown below:



2. We start by adding the edge (c, e) to the forest as it is the current lightest edge and both end points are in different trees in the forest. Our forest becomes:

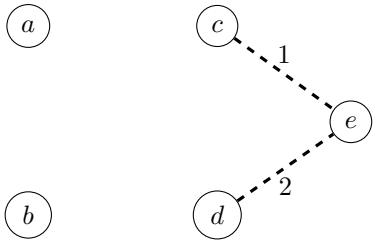


[2]I once heard, a likely apocryphal, story that it was in fact discovered by accident. I have no way to corroborate this story. Though one could see how one might accidentally happen on such an algorithm.

Our sequence is now

$$S = \langle (d,e), (d,c), (c,b), (a,c), (a,b), (b,d) \rangle$$

3. We add the edge (d,e) to the forest as it is the current lightest edge and both end points are in different trees in the forest. Our forest becomes:



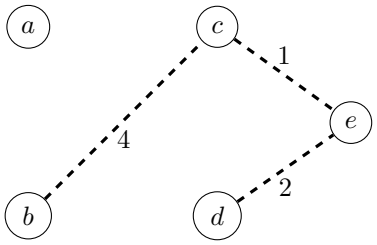
Our sequence is now

$$S = \langle (d,c), (c,b), (a,c), (a,b), (b,d) \rangle$$

4. Our next edge to process is edge (d,c) since both end points of the edge are in the same tree in the forest we don't modify the graph at all. Our sequence is now

$$S = \langle (c,b), (a,c), (a,b), (b,d) \rangle$$

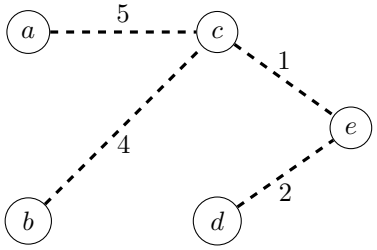
5. We add the edge (c,b) to the forest as it is the current lightest edge and both end points are in different trees in the forest. Our forest becomes:



Our sequence is now

$$S = \langle (a,c), (a,b), (b,d) \rangle$$

6. We add the edge (a,c) to the forest as it is the current lightest edge and both end points are in different trees in the forest. Our forest becomes:



Our sequence is now $S = \langle (a,b), (b,d) \rangle$. Since we only have one tree in our forest, we are done.

12.4 Dijkstra's Algorithm

- Dijkstra's[3] Algorithm is an algorithm for solving the single-source shortest-paths problem.
- Formally, the single-source shortest-paths problem is defined as:

  Definition 3 (Single-Source Shortest-Paths (SSSP) Problem)**.**
  Input*: A weighted directed graph $G = (V, E)$ and a source vertex $s \in V$*
  Output*: The set of shortest paths*

  $$\left\{ p \;\middle|\; s \overset{p}{\rightsquigarrow} v \text{ is a shortest path from } s \text{ to } v \text{ where } v \in V \right\}$$

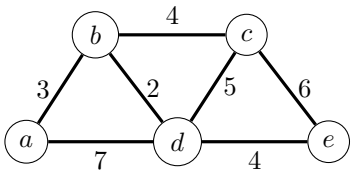
- Applications
  - transportation planning
  - Packet routing in communication networks
  - Friend discovery in social networking
    - Think friend recommendations in Facebook.
  - Speech recognition
- Dijkstra's algorithm for SSSP is perhaps the best known solution to the SSSP problem.
- Dijkstra's algorithm is applicable to
  - directed graphs with non-negative edge weights
  - undirected graphs with non-negative edge weights
- The general idea: find shortest path to the vertices of graphs in order of their distance from a given source
  - First find the the shortest path to a vertex nearest to the source
  - Proceed to find the shortest path to a vertex second nearest to the source
  - etc.
- Following the general idea we can draw the conclusion that once the i-th iteration has started, the shortest path to $i - 1$ closest vertices has been identified.
  - It turns out that this graph of paths forms a spanning tree we will denote by T_i for the $i - 1$ closest vertices.
  - Note: I said spanning tree, this is not necessarily a MST.
- The next edge nearest to the source can be found among the vertices adjacent to the to the vertices in T_i.
- If this looks like Prim's algorithm to you, you are half-correct. We again will be selecting something similar to a *light edge* where $(V, V \setminus V_i)$ defines the cut. Specifically, it will be cross-edge with lowest accumulated weight. Where $G = (V, E)$ is the original graph and $T_i = (V_i, E_i)$ is the tree.
  - The book calls the vertices adjacent to vertices in T_i *fringe* vertices.
- Book-keeping: Every vertex $v \in V$ in Dijkstra's algorithm gets two additional fields added to it
  1. The current minimum distance from the source vertex to v

[3] Pronounced dike-stra, in Dutch "j"'s are similar to the English "h"

– This field will be updated for each vertex adjacent v using a process known as *edge relaxation.*

2. The vertex in the tree that v is attached to (the predecessor).

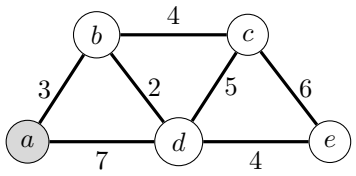
- **Practice**: What is the greedy choice in Dijkstra's algorithm?
- Let's look at an example of Dijkstras algorithm on the following graph



1. We select s to be a and have relax all outgoing edges which yields minimum priority queue

$$Q = \langle (b, 3, a), (d, 7, a), (e, \infty, \perp), (c, \infty, \perp) \rangle .$$

Which means our graph looks like

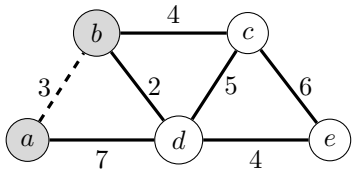


and $V_T = \{a\}$.

2. We execute an extract min operation to get $(b, 3, a)$ and relax all edges adjacent to b which means

$$Q = \langle (d, 5, b), (c, 7, b), (e, \infty, \perp) \rangle .$$

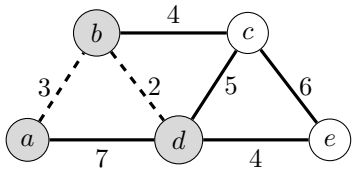
The set of tree edges is $V_T = \{a, b\}$ and the graph looks like:



3. We execute an extract min operation to get $(d, 5, b)$ and relax all edges adjacent to d which means

$$Q = \langle (c, 7, b), (e, 9, d) \rangle .$$

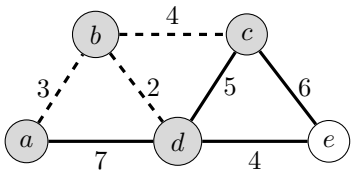
The set of tree edges is $V_T = \{a, b, d\}$ and the graph looks like:



4. We execute an extract min operation to get $(c, 7, b)$ and relax all edges adjacent to c which means

$$Q = \langle (e, 9, d) \rangle .$$

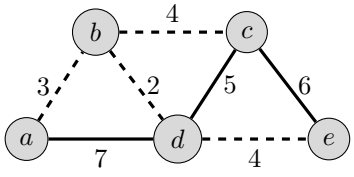
The set of tree edges is $V_T = \{a, b, d, c\}$ and the graph looks like:



5. We execute an extract min operation to get $(e, 9, d)$ and relax all edges adjacent to c which means

$$Q = \langle\rangle .$$

The set of tree edges is $V_T = \{a, b, d, c, e\}$ and the final graph looks like:



- Like Prim's algorithm the running time of Dijkstra's algorithm is also $O(|E| \lg |V|)$ if an adjacency list is used to represent the graph.

12.5 Huffman Trees and Codes

- Let's look at a different problem domain where greedy algorithms have found utility.
- The field is called information theory – in particular we will study one specific part of information theory known as compression.
- We will assume there exists an alphabet of symbols called Σ from which all of the original text uses.
  - For example, ASCII or UNICODE
- We call a sequence of bits used to represent a symbol $s \in \Sigma$ a *codeword*.
- An assignment of codeword to symbols is called an *encoding*
- There are two main types of encodings
  1. **Fixed-Length Encoding**: Each symbol is assigned a codeword of the same length
     - e.g., ASCII and UNICODE
  2. **Variable-Length Encoding**: Each symbols is assigned a codeword of possibly different lengths.
- Today we will look at variable length codes (or encodings).
- There is a large problem that we have to deal with that is special to variable length codes.
  - In general, it is hard to determine for a string of bits, representing multiple codewords, how many bits represent the codeword for a certain symbol in Σ.
    - As an example consider the alphabet $\Sigma = \{a, b, c\}$ with code words $w_a = 1, w_b = 0, w_c = 11$ and we have the sequence of bits 1011 presented to us to decode to symbols. There are two valid decodings:
      1. 1011 decoding to *abc*.
      2. 1011 decoding to *abaa*.

      This ambiguity is cause by the fact that the code is of variable length and there is no information in the concatenation of codewords that hints a unique way to split the concatenation apart.
    - We don't have this problem with fixed length codes. For example, in ASCII we know that every codeword is eight bits in length so as we read a file (stored as bytes) that every 8 bits is one codeword that corresponds to exactly one symbol.
- One way to solve the problem inherit to general variable length codes is to use a *prefix-free code*.
  - **Prefix-Free Code**: A code in which no codeword is a prefix of another codeword.
  - Prefix-free codes allow us to scan left to right through a stream of bits until we discover the first group of bits that represent a codeword.
  - Example: a prefix free code for the alphabet $\Sigma = \{a, b, c\}$ would be $w_a = 00, w_b = 01, w_c = 1$ is prefix. Do you see why?
  - **Practice**: Why is the original variable length code we gave for $\Sigma = \{a, b, c\}$ not prefix free?
- Perhaps the most famous prefix-free code is due to David Huffman in 1952[4].
- Huffman's method proceeds as follows

[4]David Huffman discovered the algorithm while he was a graduate student at MIT. It was actually a problem he solved in lieu of taking a final exam for a course. Article of the story is here `http://www.huffmancoding.com/my-uncle/scientific-american`

1. Take the source text that use a specific alphabet Σ and count the frequency of each letter in the source text.
2. Setup:
   (a) Initialize n one node trees and label them with the symbols of Σ.
   (b) Record the frequency of each symbol in its tree's root.
      – This frequency is called the tree's *weight*
      – In general the weight will be equal to the sum of the frequencies of the leaves of the tree.
3. Repeat the following until a single tree is obtained:
   (a) Find two trees T_1 and T_2 with the smallest weights (breaking ties arbitrarily)
   (b) Make T_1 the left subtree and T_2 the right subtree of a new tree with root that has weight equal to the sum of the weight of the left and right subtree.

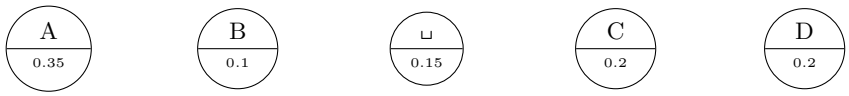
– This tree is called a *Huffman Tree*.

- **Practice**: What is the greedy choice for constructing a Huffman tree?
- From the Huffman tree we can extract codewords for each symbol that appears in a leaf of the Huffman Tree.
  – Label the left edge leaving a node with 0 and every right edge leaving a node a 1.
  – We then can describe the root-to-leaf path through the tree as a binary string.
    * This means the root-to-leaf path is a codeword for symbol stored in the leaf.
- Let's look at an example of constructing a Huffman tree from the symbol-frequency table given below

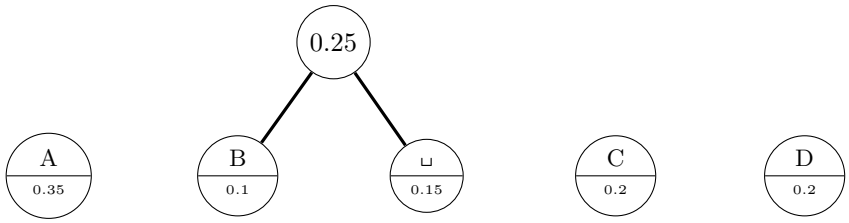
Symbol	A	B	C	D	␣
Frequency	0.35	0.1	0.2	0.2	0.15

Note we will denote a blank space by ␣.

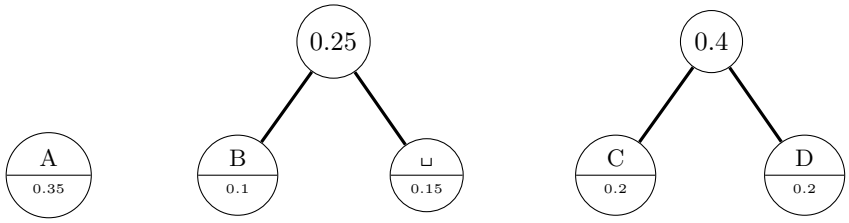
1. We start with a forest of $|\Sigma|$ trees.



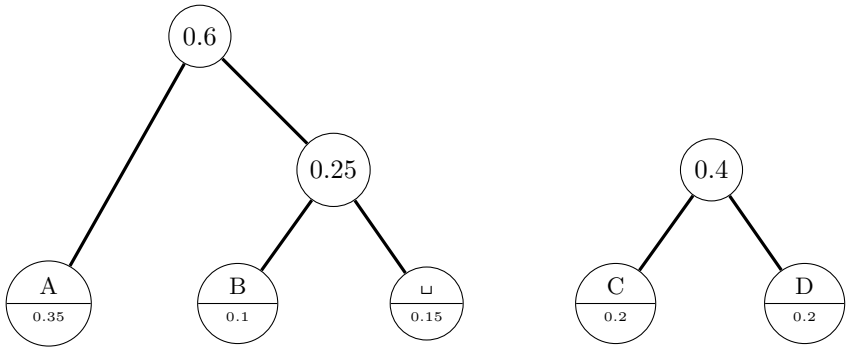
2. We first greedily join B and ␣ to form a new tree (decreasing the size of the forest by one).



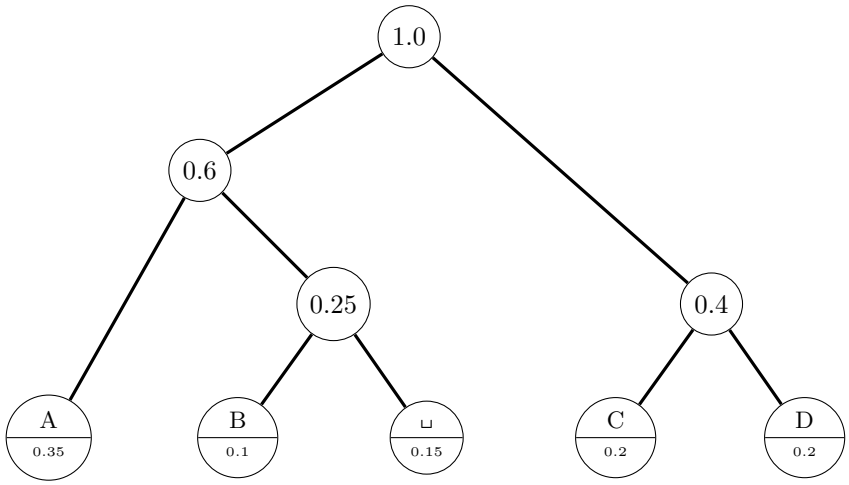
3. We proceed to join C and D to form a new tree (decreasing the size of the forest by one) as they are the current smallest two trees in the forest.



4. We proceed to join A and the tree that contains B and ␣ to form a new tree (decreasing the size of the forest by one) as they are the current smallest two trees in the forest.



5. There are only two trees remaining which we join together.



6. We then can extract the codewords for each letter and arrive at the table:

Symbol	A	B	C	D	␣
Frequency	0.35	0.1	0.2	0.2	0.15
Codeword	11	101	01	00	100

- Note: Depending on who you are you either lable the left branch with a 1 and the left branch with a 0. As long as you stay consistent you are OK.
- Notice letter with a higher frequency have shorter codewords. This is a desirable property.
- **Practice**: Encode the word "BAD" using the Huffman code above.
- **Practice**: Decode the codeword string 0011100
- We may often be concerned with the average number of bits per code word when dealing with a variable length code.
  - The formula is $\sum_{a \in \Sigma} |w_a| \, f_a$ where $|w_a|$ denotes the number bits in the codeword of a and f_a denotes the frequency of a.
  - For the example above, we have:

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25$$

- **Practice**: How many bits are needed per codeword if we use a fixed-length code to represent $\Sigma = \{A, B, C, D, \sqcup\}$?
- **Practice**: How many bits are needed per codeword if we use a fixed-length code to represent any alphabet Σ?

- While we won't talk about it a lot you may hear about compression ratio when talking about compression algorithms. The higher the number the better the compression r.
  - The compression ratio is:
    $$r = 100\left(1 - \frac{v}{f}\right),$$
    where v is the average length of a codeword in the variable length code for Σ and f is the codeword length in the fixed-length code for Σ. This is what the book gives, in fact we can make it more general.
  - The most general form of this is
    $$r = 100\left(1 - \frac{v}{\lceil \lg |\Sigma| \rceil}\right).$$

Challenge Problems

1. You are in a candy store and there are n different kinds of candy bars with costs $C_0, C_1, ..., C_{n-1}$. You have M cents to spend. Devise a greedy strategy to maximize the variety of candy you can purchase.
   - What is the time complexity of your algorithm?
   - What is the space complexity of your algorithm?
   - Why is your algorithm a greedy strategy? Justify your answer.
2. **The Assignment Problem**: The task is to assign n people to execute n jobs (one person per job). The cost for the i^{th} person to execute the j^{th} job is given by `C[i][j]`.
   - Design (in Python or simply as a list of steps) a greedy algorithm that finds a valid small-cost assignment.
   - Does this algorithm find the optimal (minimum) cost assignment? If so, why? If not, provide a counter example.
3. **Fractional Knapsack Problem:** Design a greedy algorithm to solve the "fractional Knapsack problem".
   - This is the same as the standard knapsack problem, except that each item can be broken up into pieces. So you can take any fraction of an item (or rather, any proportion at all). So you can take .45234 * i, for instance.
   - What is the running time of your algorithm?
   - What application(s) might this admittedly-absurd metaphorical problem possibly apply to?
4. Construct a greedy algorithm to solve the Traveling Salesman Problem
   - `https://en.wikipedia.org/wiki/Travelling_salesman_problem`
   - Does this algorithm find the optimal (minimum) cost assignment? If so, why? If not, provide a counter example.