3. Greedy algorithms

Attempt of a definition: An algorithm is greedy if the final result of the algorithm is retrieved via small steps, where each step aims to optimize some criterion based on information available locally at that step.

Note: It is not possible to come up with a proper definition.

Goals in the following :

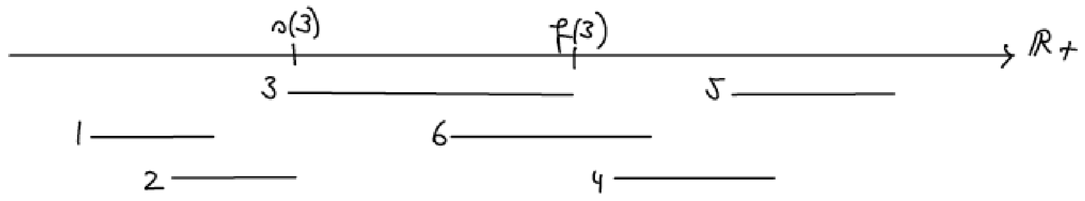
- gain an intuition for how greedy algorithms operate by considering examples
- understand how greedy algorithms can be used to derive solutions.
- examine how the efficiency of greedy algorithms compares to corresponding non-greedy ones.

3.1 Interval Scheduling (4.1) of book

Problem definition: Given a set S of n intervals, $S = \{(s(1), f(1)), ..., (s(n), f(n))\} = \{(s(i), f(i)), i \in \{1, ..., n\}, s(i), f(i) \in \mathbb{R}^+, s(i) < f(i)\}$, find an optimal subset T of S, i.e., a $T \subseteq S$ such that

- T contains no pair of overlapping intervals, i.e., for all pairs $(s(i), f(i))$ and $(s(j), f(j))$, we have either $f(i) < s(j)$ or $f(j) < s(i)$
(T is then called a compatible set of intervals)
- there is no other compatible set $T' \subseteq S, T \neq T'$, that has more elements than T, i.e., $|T'| < |T|$

visually :

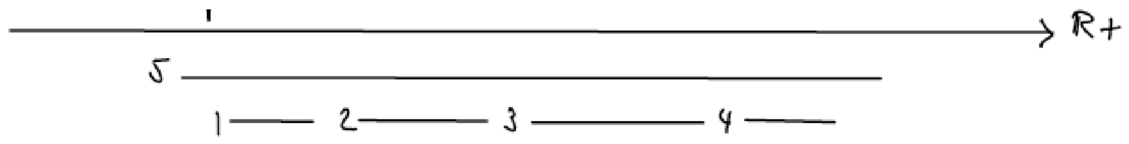


- In the above example, $T = \{(s(i), f(i)|i \in \{1, 3, 5\}\}$ is a compatible set, whereas $\tilde{T} = \{(s(i), f(i)|i \in \{1, 2, 3\}\}$ isn't because interval 2 and 3 have an overlap.

Goal: Find a greedy algorithm that solves the above problem efficiently. Group Work

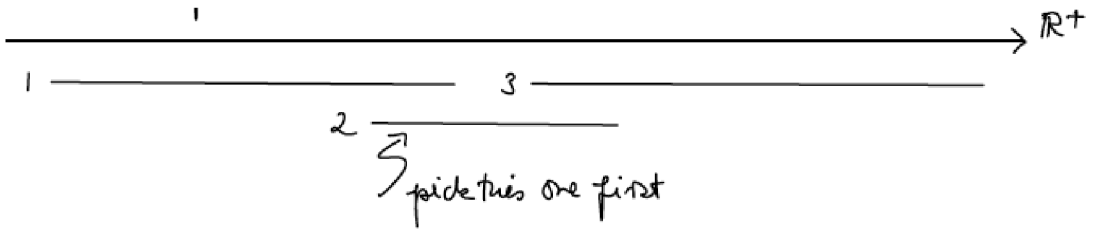
Collect ideas:

Idea 1 : Start by selecting the interval within the smallest stat value $s(i)$.
$\rightarrow$ Problem: If the first interval happens to be long, we will miss out on solutions that fit more intervals into the same spaces, i.e., using this strategy will often not retrieve an optimal subset of S.
counter example:



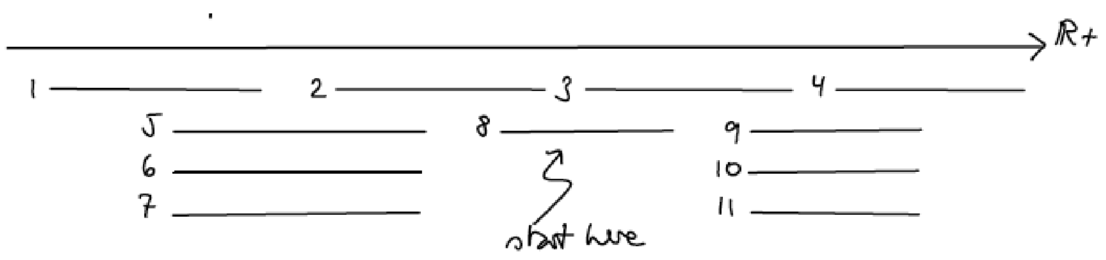
$\Rightarrow$Conclusion: We should take the length of intervals into consideration, too.

Idea 2 : Start by choosing the shortest interval, i.e., where $|f(i) - s(i)|$ is minimal.
$\rightarrow$ Problem: this idea is better than the previous, but still not guaranteed to derive an optimal subset of S
counter example:



$\Rightarrow$Conclusion: making a bad choice too early on prevents us from making better choices later on. We should take into account hot many conflicts a given interval has.

Idea 3 : Start by selecting the interval with the least number of overlapping intervals.
$\rightarrow$: this strategy does not always produce an optimal subset of S.
counterexample:



Starting with interval 8 prevents us from finding the solution $T = \{(s(i), f(i)) | i \in \{1, 2, 3, 4\}\} \subset S$.

Idea 4 : Start by picking the interval with the smallest end point f(i) to ensure that we can fit in many more intervals. This idea is similar to idea 1 and also takes interval length (idea 2) into account. If does not seem to incorporate idea 3, though, at least not in an obvious way.
We can write idea 4 in the following idea as algorithm:

Definition :

- R = set of intervals in S that we have not considered yet
- A = set of selected intervals

Algorithm : Interval scheduling algorithm Nr.1

```
– set $R = S$ and A=$\emptyset$ (the empty set)
– while R $\neq \emptyset$ {
    * choose an interval $i$ in $R$ where $f(i)$ is smallest
    * add interval $i$ to $A$
    * delete all intervals from $R$ that are incompatible with $i$ (*)
  }
– return $A$
```

(1) properties of this algorithm: correctness

Claim :The algorithm returns a compatible set of intervals $A \subset S$.

Group Work : Proof

Proof : Clear from the way the algorithm works (see * above)

Theorem : The above algorithm return an optimal subset $A \subseteq S$

Proof : We will use a proof by contradiction, i.e. we will show that we will get a contradiction if we assume that the subset $A \subseteq S$ returned by the algorithm is not optimal. So, suppose $A \subseteq S$ is not optimal. $A = \{(s(i), f(i)|i \in \{a_2, a_2, ..., a_k\}\}k \leq n$. This implies that there another compatible subset $B \subseteq S$ which contains more elements than A, i.e.

$$B = \{(s(i), f(i))|i \in \{b_1, b_2, ..., b_m\}\}, k < m \leq n$$

In order to continue, we need the following lemma.

Lemma : Let the sequence of interval indices in A, $a_1, a_2, ..., a_k$ corresponding to the orders in which the corresponding intervals were added to A. Order the indices of intervals in B, $b_1, b_2, ..., b_k$ according to the natural left-to-right order of the corresponding intervals. Note that as A and B are compatible set, there intervals can be simultaneously ordered by their start and end points.
For all indices $r \leq p \Rightarrow f(a_r) \leq f(b_p)$, where $r \in \{a_1, a_2, ..., a_k\}$ and $p \in \{b_1, b_2, ..., b_m\}$

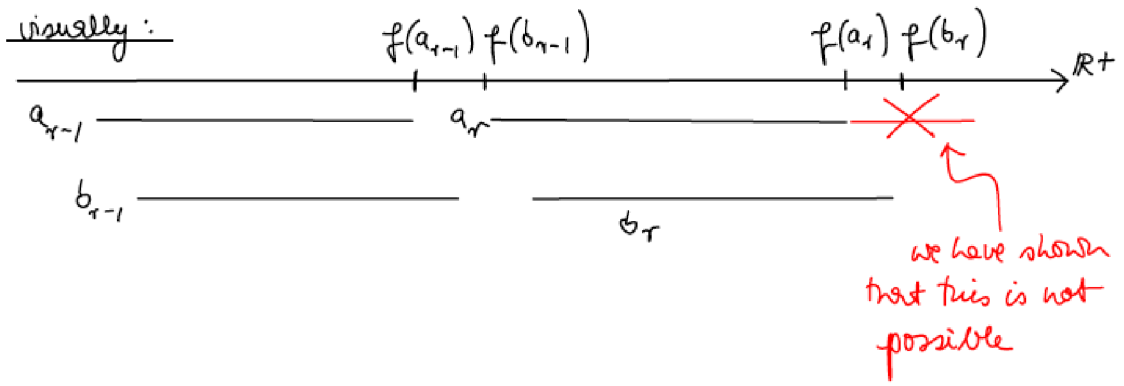
Proof :
We will use a proof by induction.
Induction start:
r = 1: The statement $f(a_1) \leq f(b_p)$, for all $p \in \{b_1, b_2, ..., b_m\}$ is true as the above algorithm starts with the interval in S with the smallest end-point.
Induction Step (r-1→r):
r >1: We assume that the statement is correct for (r-1) and will try to prove that this implies the correctness of the statement for r.
Given $f(a_{r-1}) \leq f(b_{r-1})$ and given that B is a compatible set whose indices are ordered as described above, i.e. $f(b_{r-1}) \leq s(b_r)$, We get

$$f(a_{r-1}) \leq s(b_r)$$

Because of this, interval $(s(b_r), f(b_r))$ is therefore in the set R of intervals that are still available when the greedy algorithm above select interval $(s(a_r), f(a_r))$. As the greedy algorithm selects the available interval with the smallest end-point, we have:

$$f(a_r) \leq f(b_r)$$

Conclusion: For every r-th interval selected by the above greedy algorithm for subset A, the end-point of this interval $(s(a_r), f(a_r))$ is to the left (or exactly at) the end-point of the r-th interval in B.



Proof of the last theorem (continued) :
Keep order of intervals in A and B as in proof of lemma.

Remember, we assumed k<m. According to the just-proven lemma, this implies:
$f(a_k) \leq f(b_k) < s(b_{k+1})$
This implies that set R is the algorithm was <u>not</u> empty when the algorithm terminated and returned as answer subset A. This is a contradiction to our conditions defined at the start of the proof. It then follows $m \leq k$, i.e. A is an optimal subset of S.

(2) <u>properties of this algorithm: worst-case running time</u>

Claim 1 :The above interval scheduling algorithm has a worst-case running time of O(n^2).

Proof : Group Work

- the while loop requires O(n) steps and the statements within the while loop require at most O(n) steps, i.e. O(n^2)

Claim 2 :We can come up with an algorithm that solves the interval scheduling problem in at most O(nlog(n)) time.

Proof : Group Work: Can you think of ways of improving the above algorithm?

Algorithm : Interval Scheduling algorithm Nr.2

```
• definitions as in the algorithm above
• xmax = $-\infty$
• set A = $\emptyset$ and R = S
• sort R by increasing finishing time (*) $\rightarrow$ ( takes O(nlog(n)) )
• for each interval i in R { $\rightarrow$ ( takes O(n) )
    – if s(i) > xmax { (*)
        ∗ add i to A $\rightarrow$ ( takes O(1) )
        ∗ xmax = f(i) $\rightarrow$ ( takes O(1) )
    – }
• }
• return A
```

Note : The clever combination of the two steps (*) above avoids us having to remove intervals that are incompatible with interval i.
$\Rightarrow$ conclusion: the worst-case time requirement of this algorithm is O(nlog(n)).

Motivation:

in the following examine several greedy algorithm and show

- these are correct, i.e. do what they claim to achieve and
- how they compare to non greedy algorithms, e.g. in terms of efficiency

3.2 Applications of greedy algorithms

3.2.1 Data compression

Goal: given a sting X over alphabet S, we want to replace each character in the string by a sequence of bits so that

- the length of the resulting string is as short as possible
- we can recover the original string from the string of bits

example: $X = abacbaada \quad S = \{a, b, c, d\}$

$a \to 00$

$b \to 01$

$c \to 10$

$d \to 11$

using this mapping string X needs 000100100100001100, i.e. it is 18 bits long.

Definition: A fixed-length code maps all characters from the alphabet S to strings of bits of the same fixed length.

Group Work: Given an alphabet S of $k = |S|$ different characters. How long does the fixed-length code have to be to encode this?

Answer: $\lceil \log_2(k) \rceil$ because we can encode $2^m = k$ character sequences with m bits.

Definition: A variable-length code maps characters from the alphabet S to string of bits of possibly different lengths.

example: As in the example above, $X = abacbaada \quad S = \{a, b, c, d\}$

$a \to 0$

$b \to 10$

$c \to 110$

$d \to 111$

using this code, X needs 010011010001110, i.e. it is 15 bit long.

Advantage: Using this code, we use few bits for the most common character in the string (a). Overall we need fewer bits even though some characters require more than 2 bits as in the example above.

How about the decoding? Can this be done unambiguously?
0|10|0|110|10|0|0|111|0

$\overset{\downarrow}{a}\ \overset{\downarrow}{b}\ \overset{\downarrow}{a}\ \overset{\downarrow}{c}\ \ \overset{\downarrow}{b}\ \overset{\downarrow}{a}\,\overset{\downarrow}{a}\ \ \overset{\downarrow}{d}\ \ \overset{\downarrow}{a} \longrightarrow X = abacbaada$ OK$\surd$

Read in $\overrightarrow{\text{one}}$ direction

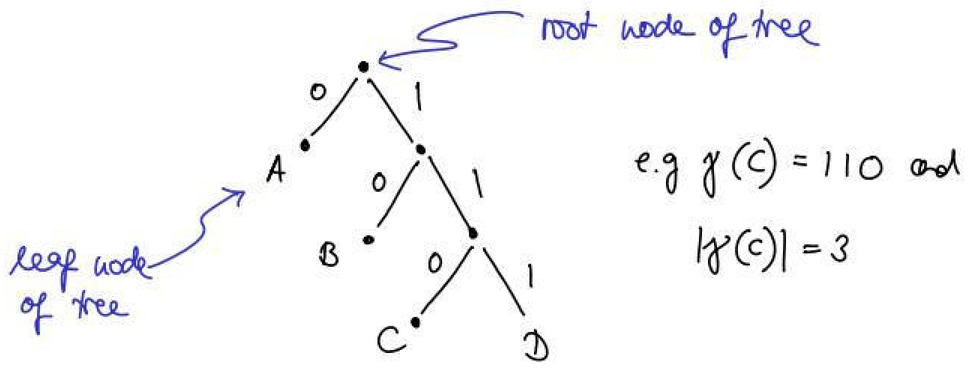
Motivation: Decoding becomes ambiguous if the bit string that encodes one character is the prefix of the bit string of another character.

Definition: A prefix code for an alphabet S is a function γ that maps every element from S to a bit string in such a way that for all $x, y \in S$, the bit string $\gamma(x)$ is not a prefix of $\gamma(y)$, $x \neq y$.

Decoding using a prefix code: Given a string X of characters from alphabet S and a prefix code γ, read characters from start of X until the bit string of $\gamma(x)$ of a $x \in S$ is matched, translate this into x and continue decoding the rest of X in this way until the end of X is reached.

Visualization: A prefix tree can be represented by a binary tree whose

- every non-leaf node has two child nodes (hence binary tree)
- edges are labeled 0 or 1
- the leaves are the letters from the alphabet and
- the sequence of bits encountered when going from the root node to the leaf node corresponds to $\gamma(x)$, where $x \in S$ is at the leaf node.



Motivation: How do we find a prefix code that is optimal for a given text?

In the following assume: Given string X of characters from alphabet S that is to be encoded, we are allowed to read the text <u>before</u> deciding on the bit encoding. In particular, we are allowed to measure the frequency $f(x) \in [0, 1]$ of every character $x \in S$ in the string X.

Goal: find a prefix code γ for which

$$L_T := \sum_{i=1}^{|S|} f(c_i) \cdot |\gamma(c_i)|$$

is minimal.
L_t = total length of X encoded using γ
$\gamma(c_i)$ = length of bitstring for c_i, equal to depth of binary tree, i.e. $|\gamma(c)| = |110| = 3$.

Observation: Frequently used letters should be encoded with few bits, i.e. be at the top of the encoding binary tree.

Huffman's algorithm to construct an optimal pre-fix code

given a string X of characters from the alphabet S and their frequency $f(x), \forall x \in X$

Idea: Construct the tree corresponding to the optimal pre-fix code in a recursive fashion. Starting at the leaves of the tree, combine the two nodes with the lowest frequency into a new node.

Algorithm:

- if S with $|S| = 2$, then use 0 to encode one letter and 1 to encode the other letter
- else {
  - let y^* and z^* be the two letters in the alphabet S with the lowest frequency
  - define a new alphabet S' by deleting y^* and z^* from S and replacing them with a new letter w of frequency $f(y^*) + f(z^*)$
  - recursively construct a prefix code γ' for S' with binary tree T'
  - define a prefix code for S as follows:
    * start with T'
    * take the leaf labeled w and add the corresponding two children y^* and z^* as child nodes

  }

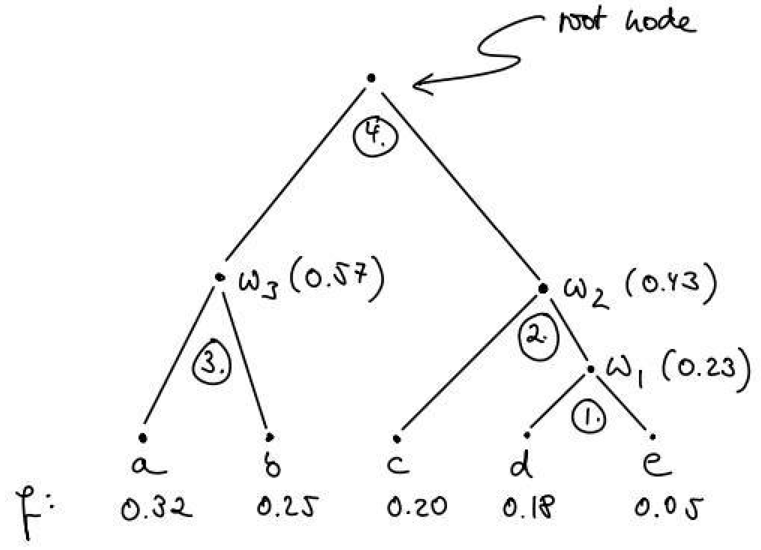
Definition: The prefix code constructed by Huffman's algorithm is called Huffman's Code.

Example: $S = \{a, b, c, d, e\}$ and some string X such that:
$f(a) = 0.32$, $f(b) = 0.25$, $f(c) = 0.20$, $f(d) = 0.18$, $f(e) = 0.05$

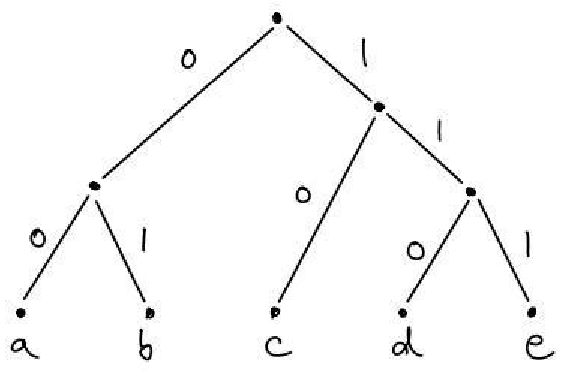
Group work: Construct the corresponding Huffman's Code based on the above algorithm.

apply the Huffman's algorithm:

1. merge d and e to w_1 with $f(w_1) = 0.23 \Longrightarrow S_1 = \{a, b, c, (de)\}$
2. merge w_1 and c to w_2 with $f(w_2) = 0.43 \Longrightarrow S_2 = \{a, b, (cde)\}$
3. merge a and b to w_3 with $f(w_3) = 0.57 \Longrightarrow S_3 = \{(ab), (cde)\}$
4. as $|S_3| = 2$, invoke if-clause of algorithm and we are done



$\Longrightarrow$ The resulting tree with the edges below each node, labeled by 0's and 1's then corresponds to a prefix code, i.e. Huffman's code.



Question: Is this prefix code optimal? i.e. is $\sum_{i=1}^{|S|} f(x_i) \cdot |\gamma(x_i)|$ minimized?

Group work: Is the optimal prefix code unique?

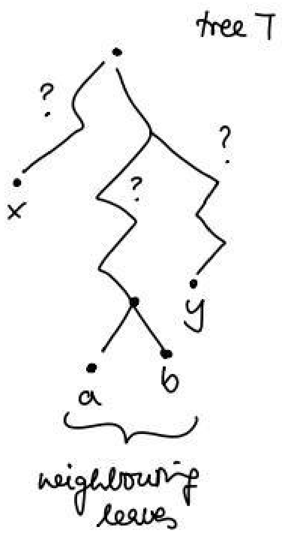
Answer: No. Can label the two edges underneath each node 0 and 1 rather than 1 and 0.

Claim 1: There is an optimal prefix code in which the two letters with the lowest frequency have code of the same length that differs only in the last bit.

Def. The length of a prefix code γ for string X and alphabet S whose elements x_i have frequency $f(x_1)$ in string X is defined as $\sum_{i=1}^{|S|} f(x_i) \cdot |\gamma(x_i)|$

Proof: Suppose we have an optimal prefix code with the corresponding tree T. Let x and y be the two characters with the lowest frequency. We now show how to make x and y neighbouring leaves without changing the maximum length of the code.
Let a and b be two characters at two currently neighbouring leaves of T. Assume WLOG (without loss of generality) that

$$\begin{aligned} f(a) &\leq f(b) \quad \text{and} \\ f(x) &\leq f(y) \quad \text{with } a \neq y \text{ and } b \neq x \end{aligned}$$



Idea: Go from tree T to $T^{'}$ by swapping $x \leftrightarrow a$ and $y \leftrightarrow b$ to make x and y the neighbouring leaves in $T^{'}$ and show that length$(T) >$ length$(T^{'})$.

Define: For a tree T representing a prefix code γ for string X and alphabet S whose elements x_i have frequency $f(x_1)$ in string X define length$(T) = \sum_{i=1}^{|S|} f(x_i) \cdot |\gamma(x_i)|$

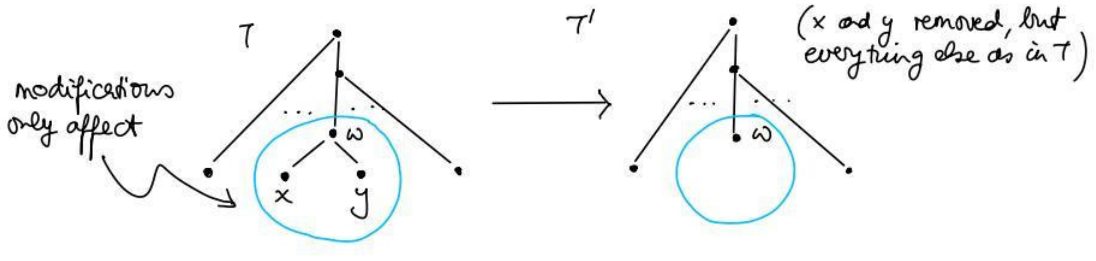
$$\begin{aligned} \text{length}(T) - \text{length}(T^{'}) &= |\gamma_T(x)|f(x) + |\gamma_T(a)|f(a) - (|\gamma_{T'}(x)|f(x) + |\gamma_{T'}(a)|f(a)) \\ &= f(x)(|\gamma_T(x)| - |\gamma_{T'}(x)|) + f(a)(|\gamma_T(a)| - |\gamma_{T'}(a)|) \\ &= f(x)(-\alpha) + f(a)(\alpha) \\ &= \alpha \underbrace{(f(a) - f(x))}_{\downarrow} \end{aligned}$$

> 0 because x has $f(x) \leq f(a), \forall a \in S, a \neq y$.

we can show the same when swapping $y \leftrightarrow b$, provided $b \neq x$.
$\Longrightarrow T^{'}$ is another optimal prefix tree with the claimed property $\square$.

Claim 2: Given a prefix tree T that satisfies claim 1, where the two letters x and y with lowest frequency are neighbouring leaves. If we remove the leaves of x and y and replace their parent node by w with frequency $f(w) = f(x) + f(y)$, then we obtain a new tree $T^{'}$ and alphabet $S^{'}$ for which length($T^{'}$) = length(T) - $f(w)$.

Proof: Visually fairly clear:


T
modifications only affect
w
x
y
T'
(x and y removed, but everything else as in T)
w

Proof:

$$
\begin{aligned}
\text{length}(T) &= \sum_{c\in S} f(c)\cdot|\gamma_T(c)| \\
&= \left(\sum_{c\in S,\, c\notin\{x,y\}} f(c)\cdot\underbrace{|\gamma_T(c)|}_{=|\gamma_{T'}(c)|}\right) + f(x)\cdot|\gamma_T(x)| + f(y)\cdot|\gamma_T(y)| \\
\left.\begin{array}{r} |\gamma_T(x)| = |\gamma_T(y)| = \\ |\gamma_T(w)|+1 = \\ |\gamma_{T'}(w)|+1 \end{array}\right\} &= \left(\sum_{c\in S,\, c\notin\{x,y\}} f(c)\cdot|\gamma_{T'}(c)|\right) + \underbrace{(f(x)+f(y))}_{=f(w)}\cdot\overbrace{(|\gamma_{T'}(x)|+1)}^{\text{merge with sum}} \\
&= \left(\underbrace{\sum_{c\in S'} f(c)\cdot|\gamma_{T'}(c)|}_{=\text{length}(T')}\right) + f(w) = \text{length}(T^{'}) + f(w) \quad \Box
\end{aligned}
$$

Theorem: Huffman's algorithm generates an optimal prefix code.

Proof. We will use a proof by induction.
Induction start:
$|S| = 2$: that is S contains two letters. This means the optimal encoding assigns a single bit (that is 0 or 1) to each of the letters, which is exactly what Huffman's algorithm does.
Induction step ($|S|-1 \to |S|$):
$|S| > 2$: We assume that Huffman's algorithm returns an optimal prefix tree for alphabets of size $|S|-1$, and we prove this implies Huffman's algorithm returns an optimal

prefix tree for any alphabet of size $|S|$.
To do this we we use a proof by contradiction. Suppose Huffman's algorithm produces a suboptimal prefix code, with corresponding prefix tree T, when run on S. There is an optimal tree Z such that length$(Z) <$ length(T), with the two letter x and y with lowest frequency as siblings by claim 1.
Replace the parent node of x and y with w such that $f(w) = f(x) + f(y)$ in Z, T, and in the alphabet S. Let Z', T', and S' be the trees and alphabet after the replacement. Note that T' is the tree that Huffman's algorithm generates when run on S', and thus must be optimal.
By claim 2:
length$(T'))$ = length$(T) - f(w)$
and length(Z') = length$(Z) - f(w)$
Since length$(Z) <$ length(T), length$(Z') <$ length(T'), which contradicts the optimality of T'.
Therefore T must be optimal, and hence Huffman's algorithm generates an optimal prefix tree for any given alphabet S. □

3.2.2 Finding shortest paths in graph

Motivation:

Many networks in which one travels from one point to another (e.g. traveling on highways via interchange, sending a signal through cables and connecting nodes) can be interpreted mathematically as graphs where the nodes of the graph represent the nodes of the network and where the edges between nodes represent the highway/cable/etc. linking the nodes of the network. When dealing with networks, we often ask questions such as:

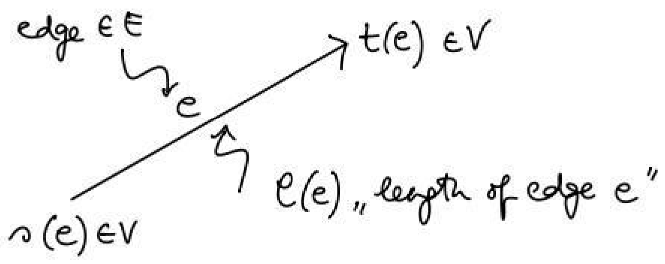
- what is the shortest path linking two nodes in the network?

  assign weights to edges in the corresponding graph which correspond to the length of the corresponding highway/cable in the corresponding network.

- **Note:** Some networks may correspond to directed graphs (if the links in the underlying network can only be traveled in one direction), whereas others may correspond to undirected graphs.

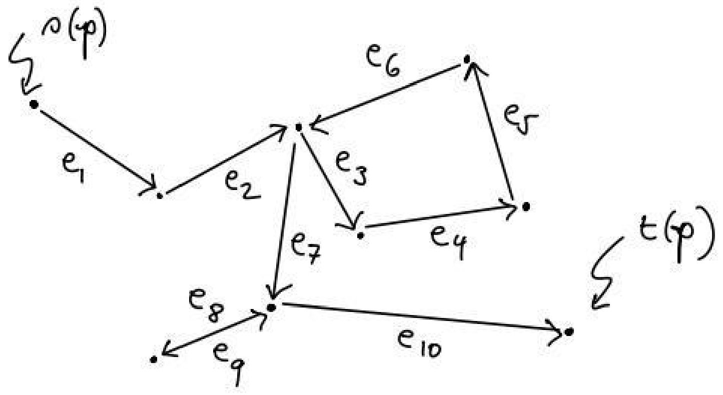
Assumption: Assume for the rest of this section that we are dealing with directed graphs.

Definition: A graph $\mathcal{G} = (V, E, s, t)$ consists of two sets V (set of verticies or nodes) and E(set of edges) and two maps $s : E \mapsto V$ (source) and $t : E \mapsto V$ (target).
A graph $\mathcal{G}$ is called finite if V and E are finite.
A Weighted graph $(\mathcal{G}, e)$ is a graph $\mathcal{G} = (V, E, s, t)$ with a function e: $E \mapsto \mathbb{R}_+$ (called the length).



Two nodes a and b are called adjacent if there is an edge e connecting them.
A path p of $m \in \mathbb{N}, m > 0$, edges in graph $\mathcal{G}$ is an m-tuple $p = (e_1, \ldots, e_m)$ of edges $e_i \in E$ such that $t(e_j) = s(e_{j+1}), \forall j \in \{1, \ldots, m-1\}$. The source and target of p are defined as $s(p) = s(e_1)$ and $t(p) = t(e_m)$ as the path p is said to start in $s(p)$ and finish in $t(p)$.

Example: Path of 10 edges whose length is $l(p) = \sum_{i=1}^{10} l(e_i)$.



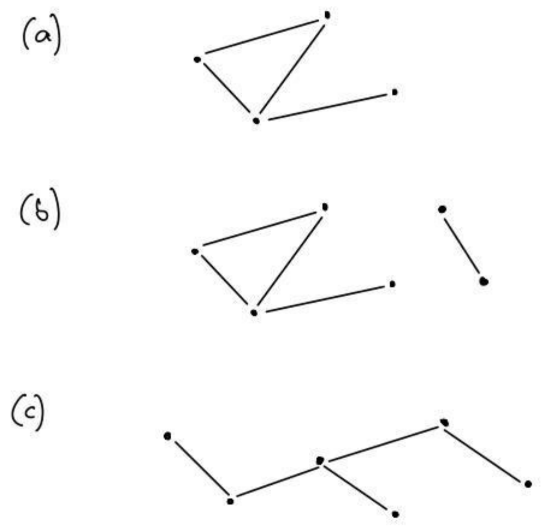
A path of 0 edges is just a vertex $p \in V$ with $s(p) = t(p) = p$.
The set of paths of $m \in \mathbb{N}$ edges in $\mathcal{G}$ is called $\mathcal{G}^{(m)}$.
If $\mathcal{G}$ is a weighted graph, $l(p) = \sum_{i=1}^{m} l(e_i)$ is called the length of path p.
A simple path is a path where every node occurs at most once (the above example is therefore not a simple path).
A simple cycle is a path that starts and ends at the same node and whose every edge appears at most once and where no node apart from the star/end node appears more that once.
A graph is called undirected, if for every edge e from node a to node b corresponds to a unique edge from b to a of the same length, otherwise the graph is a directed graph.
An undirected graph is called a connected graph if there is a path between any two nodes in the graph.
An undirected, connected graph is called a tree if it does not contain any cycle.

Group work:

(a) draw an undirected graph.

(b) draw an undirected graph which is not connected.

(c) draw a tree.



Data structures for graphs

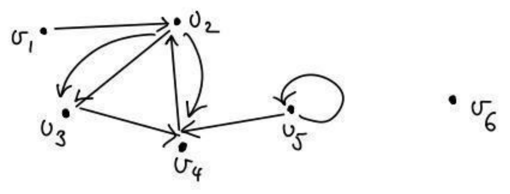
Motivation: Require data structures to store information about a graph. Data structures should be such that we can easily check properties of the graph such as

- Are two nodes a and b adjacent?

Definition: The adjacency matrix of a graph $\mathcal{G}$ is a square matrix $A = (a_{ij})$ of size $|V| \times |V|$ where matrix entry a_{ij} (i : row, j: column) is a number $n \in \mathbb{N}_0$ which corresponds to the number of edges from node v_i to node v_j.

Group work:

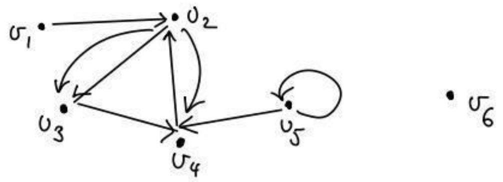
(a) Take this directed graph and specify its adjacency matrix A.



(b) Given an adjacency matrix A, how can you tell if it corresponds to an undirected graph?

- The matrix is symmetric matrix, i.e. $a_{ij} = a_{ji}, \forall i, j \in \{1, \dots, |V|\}$.

(a) answer:



$$A = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 2 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Features of adjacency matricies: Group work

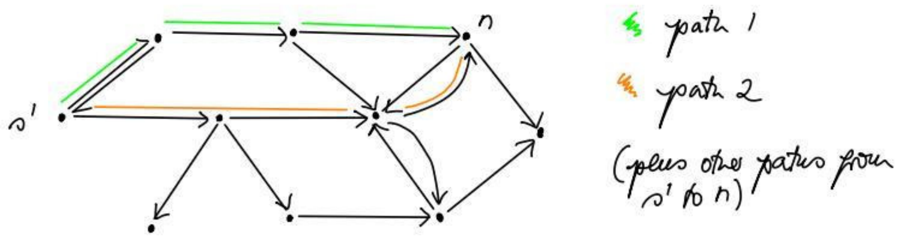
(+) How much time does it take to check if two given nodes v_i and v_j are adjacent?
   - Need to only look at a_{ij} and a_{ji}.

(+) Almost all the information on a graph is stored in the matrix which requires $O(|V| \times |V|)$ memory to be stored.
   - What is missing?
   - Information on length of each edge is missing.

(-) For sparsely connected graphs storing the information in a square adjacency matrix may be memory-wise sub-optimal (as most entries are zero).

An algorithm for finding the shortest path in a graph: Dijkstra's algorithm (1959)

Goal: Given a directed, weighted graph $\mathcal{G} = (V, E, s, t)$ with length function l and a node (so called start node) s' in the graph that has a path to every other node in $\mathcal{G}$. Determine the shortest path fro s' to every other node in the graph.

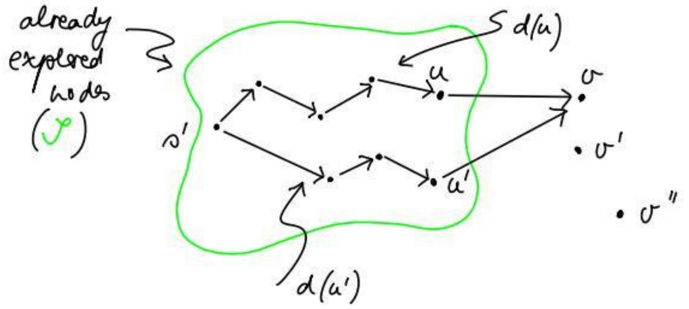


Key ideas and strategy:

- first, find an algorithm to determine the <u>length</u> of the shortest path between s' and a node n.
- second, determine the corresponding path itself which links s' to n.
- ideas for the first step
  - explore the graph by starting at start node s'
  - keep track of subset of nodes $\mathcal{S} \subseteq V$ for which we know the shortest-path distance $d(u)$ from s'. (This set $\mathcal{S}$ corresponds to the already explored part of graph $\mathcal{G}$)
  - at the start:

$$\mathcal{S} = \left\{s'\right\} \text{ and } d(s') = 0 \quad \text{(we are at the start node } s'\text{)}$$

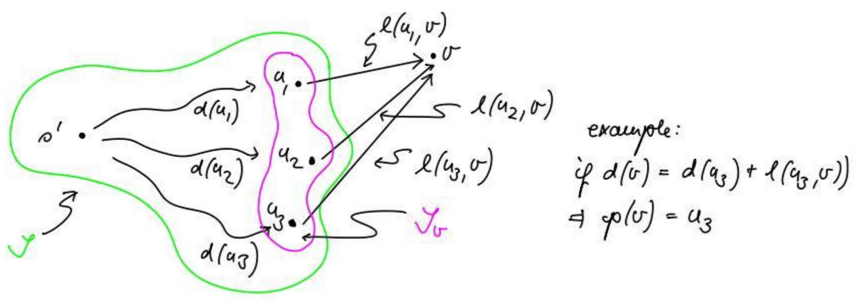
  - next,
    - for each node $v \in V \setminus \mathcal{S}$, i.e. every node that we have not explored yet, determine the <u>shortest</u> path of traveling within $\mathcal{S}$ from s' to a node $u \in \mathcal{S}$, followed by a <u>single</u> edge from u to v



  - for each $v \in V \setminus \mathcal{S}$, consider all nodes $u \in \mathcal{S}$ which have a single edge (u, v) (we call this set $\mathcal{S}_v$) and set

$$\begin{aligned} d(v) &= \min_{u \in \mathcal{S}_v} \{d(u) + l((u,v))\} \\ \text{pointer} \rightarrow p(v) &= \underbrace{\arg\min_{u \in \mathcal{S}_v}}_{\Downarrow} \{d(u) + l((u,v))\} \in \mathcal{S}_v \end{aligned}$$

i.e. the node $u \in \mathcal{S}_v$ that minimizes the term in the bracket



  - we then merge v with $\mathcal{S}$, i.e. it becomes part of the explored part of the graph and use $d(v)$ as its shortest path distance to $s^{'}$.

We can write the first step more formally as follows as Dijkstra's Algorithm:

- let $\mathcal{S}$ denote the set of explored nodes in directed, weighted graph $\mathcal{G} = (V, E, s, t)$ with length function l and start node $s^{'}$ which has a path to every node $u \in V \backslash \{s^{'}\}$
  - for each node $u \in \mathcal{S}$, we store a distance $d(u)$
- initially $\mathcal{S} = \{s^{'}\}$, where $s^{'}$ is the start node and $d(s^{'}) = 0$
- while $\mathcal{S} \subsetneq V$ {
  - select a node $v \in V \setminus \mathcal{S}$ which is connected by at least one edge to a node in $\mathcal{S}$ and where the total distance $d(v) = \min_{u \in \mathcal{S}_v} \{d(u) + l((u, v))\}$ is minimized, and set $p(v) = \arg\min_{u \in \mathcal{S}_v} \{d(u) + l((u, v))\}$
  - merge v and $\mathcal{S}$ and assign it $d(v)$.

  }
- Dijkstra's algorithm determines the shortest path distance from a start nod $s^{'}$ to any node $v \in V$.

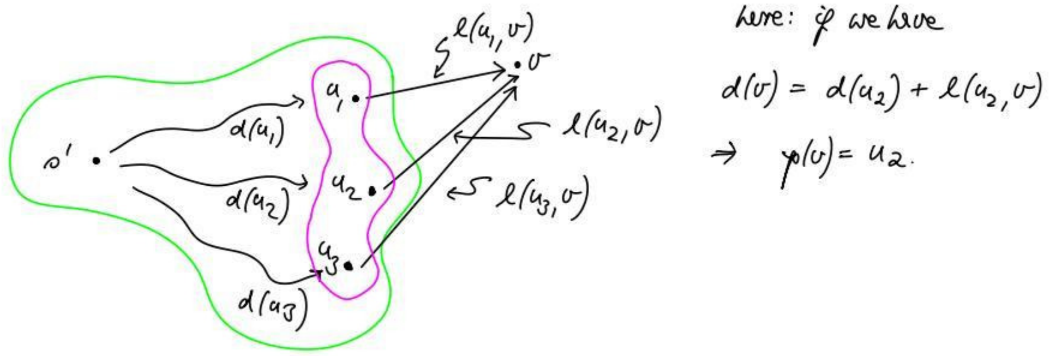
Group work: How do we obtain the corresponding shortest path between $s^{'}$ and any other node $v \in V$, i.e. how do we solve the second part of the task?

Answer: we can determine $d(v)$ in Dijkstra's algorithm using

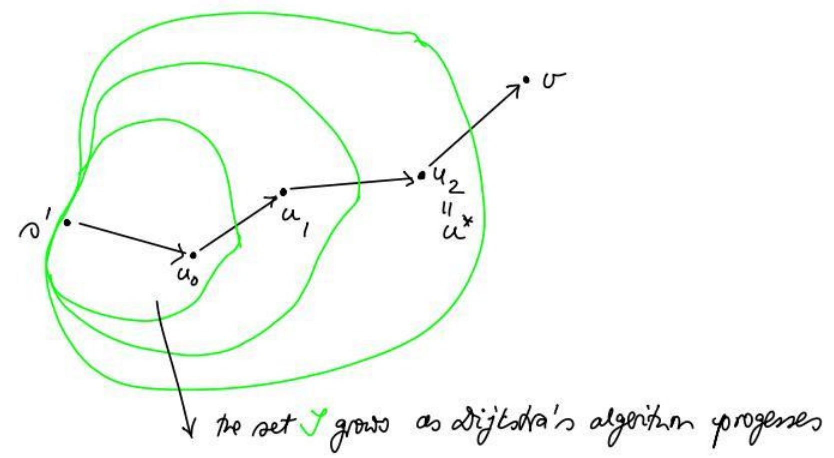
$$d(v) = \min_{u \in \mathcal{S}_v} \{d(u) + l((u, v))\}$$

we keep track of the previous node by string the pointer

$$p(v) = \arg\min_{u \in \mathcal{S}_v} \{d(u) + l((u, v))\} = \text{ the node } u \in \mathcal{S}_v \text{ that minimizes } d(v)$$



If we keep track of all these pointers at every minimization step for a given node $v \in V, v \neq s^{'}$, then we get the shrtest path p_v between start node $s^{'}$ and v by reversing the order of these nodes and connecting them by the corresponding edges.



here $p_v = \left(\left(s^{'}, u_0\right), (u_0, u_1), (u_1, u_2), (u_2, v)\right)$ because $p(v) = u_2, p(u_2) = u_1, p(u_1) = u_0$ and $p(u_0) = s^{'}$

Observations from Dijkstra's algorithm(above):

- Every iteration of the while-loop adds one node to T and decreases the set $V|T$ by one. i.e. the while loop executes at most $|V| - 1$ times (s' is not considered)

Correctness of Dijkstra's algorithm:

Reminder: Dijkstra's algorithm finds the shortest-distance $d(v)$ for any given node $v \in V|\{s'\}$, where s' is the start node w.r.t. the distance is measured.

Group Work: How would you go about the proof of the algorithm's correctness?

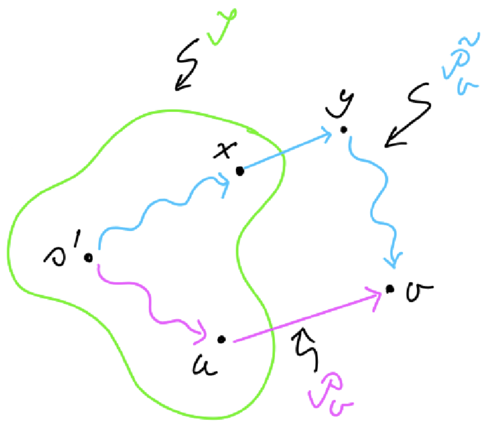
Proof: The idea for this proof is to use a proof by induction on the size of T.

Induction start: $|T| = 1$ because at the start $T = s'$ and $d(s') = 0$. The claim holds for $|T| = 1$.

Induction step: $k = k + 1$

Suppose the claim holds for $|T| = k, k \geq 1$. Show that this implies the correctness of the claim for $k + 1$ when we add a new node v to T. Let (u, v) be the final edge on the corresponding path P_v.

We know from the induction hypothesis (i.e. the correctness of the claim for $|T| = k$) that P_u is the shortest path for every $u \in T$ and $d(u)$ is the shortest distance from s' to $u \in T$.



Idea: We now consider any other path $\widetilde{P_v}$ from s' to v and show that it must be at least as long as P_v. In order to reach v, the path $\widetilde{P_v}$ must leave T somewhere. Define:

y = first node in $\widetilde{P_v}$ that is not in T

x = node just before y in $\widetilde{P_v}$, i.e. $x \in T$

Suppose $\widetilde{P_v}$ was shorter than P_v. In that case, the algorithm would have added node y to T via edge (x, y) in iteration $k+1$ rather than adding noe v. As this did not happen, it implies that

$$d(y) > d(v)$$

which implies that the length of $\widetilde{P_v}$ is greater than P_v. □

Observations

1. The while loop takes $(n-1)$ iterations for a graph with $|V| = n$ nodes because each iteration moves a node $V|\{s'\}$ to T.

2. In each iteration of the while loop, how do we efficiently select the node $v \in V|T$ to be added next to T?

   - Remember the values $d(v) = \min_{u \in T_{k-1}}\{d(u) + l((u,v))\}$ for every $v \in V|T$ at any time.
   - And order the nodes in $V|T$ in a heap-based priority queue where the distances $d(v)$ serve as keys.
   - If v' is the winning node in iteration k, change $T_k = T_{k-1} \bigcup v'$ and remove node v' and $d(v')$ from the priority queue.
   - In the next iteration $k+1$, consider all nodes $v \in V|T_k$:

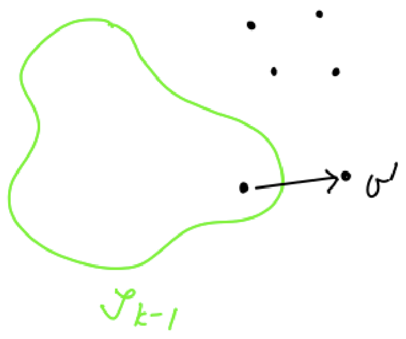
     If $(v', v) \in E$, i.e. if there is an edge from v' to v, then we may need to assign a new value to $d(v)$ by computing (index k denotes the iteration):

$$d_{k+1}(v) = \min\{d_k(v), d_k(v') + l((v', v))\}$$

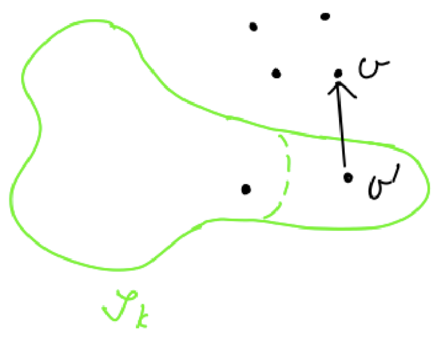
otherwise

$$d_{k+1}(v) = d_k(v) \text{ (i.e. no change is required)}$$

iteration k: add v'



iteration $k+1$: (case (*))



Group Work: Why do we only need to consider v' in the first equation rather than all $v' \in S_k$?

If $d_k(v') + l((v', v)) < d_k(v)$, then we need to update the corresponding key in the priority queue.

3. Throughout the algorithm, we need to change a key at most for every edge in the graph, whenever the node at the end of that edge is added to T.

4. As we have at most $|V| = n$ elements in the priority queue at any time, we need at most $O(\log(n))$ time to

   - insert or delete an element,
   - change an element's key, or
   - extract the element with the minimum key

   in the heap-based priority queue.

Worst-case time requirements of Dijkstra's algorithm:

For a graph with $|V| = n$ nodes and $|E| = m$ edges, the algorithm requires at most $O((m+n) \cdot \log(n))$ time.

Proof:

- We require at most $m \cdot \log(n)$ to update at most m keys in the heap-based priority queue which requires $\log(n)$ each (See observations 3 and 4).
- We require at most $n \cdot \log(n)$ to delete at most n elements from the heap-based priority queue which requires $\log(n)$ each (See observations 1, 2, and 4).

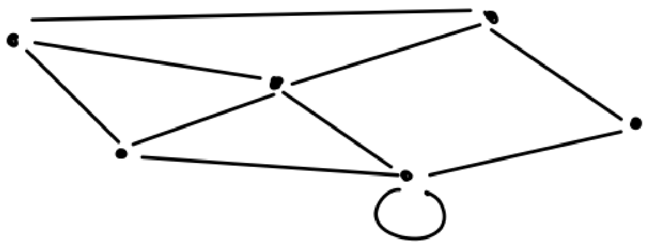
3.2.3 Minimum Spanning Trees

Motivation Wish to build a communication network whose signals can travel in both directions given a link (i.e. corresponding graph is undirected) and where the overall cost of the network in terms of the sum of all edge lengths in the graph is minimized. The nodes in the communication network (e.g. houses) all have to be reached.

Reminder An undirected, connected graph is called a tree if it does not contain any cycle.

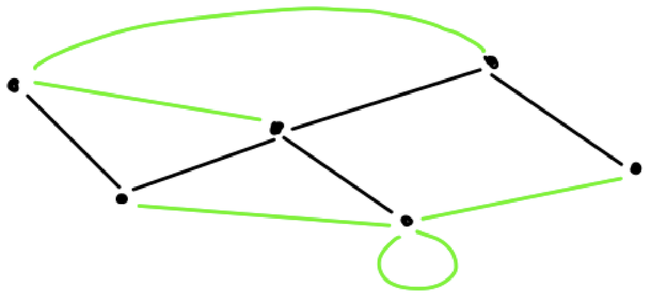
Definition A spanning tree in a graph G is a tree in G that contains every node in G.

Group Work Determine a spanning tree for the following graph G:



Answer:

(Black corresponds to belonging in G and the spanning tree of G, and green responds to only belonging to the graph G)



Goal Identify a spanning tree T whose edges have a minimum total edge length, i.e. where

$$cost(T) := \sum_{\text{e is edge in T}} l(e) \text{ is minimized}$$

Call this tree a minimum spanning tree (MSP).

Kruskal's Algorithm for Identifying a Minimum Spanning Tree

- Given an undirected, connected and weighted graph $G = (V, E, s, t, l)$, where, $l : E \to \mathbb{R}+$ denotes the length function (we will also call this function a weight-function in this function).

Group Work How would you go about finding a minimum spanning tree?

Kruskal's Algorithm

- start with a subgraph G' which contains only the isolated nodes from G (no edges yet)
- while the number of edges in G' is $< |V| - 1$ {
  - of all the edges in G that have not yet been added to G', pick the edge e with the lowest value $l(e)$,
  - add this edge to G' unless it creates a cycle in G'

  }

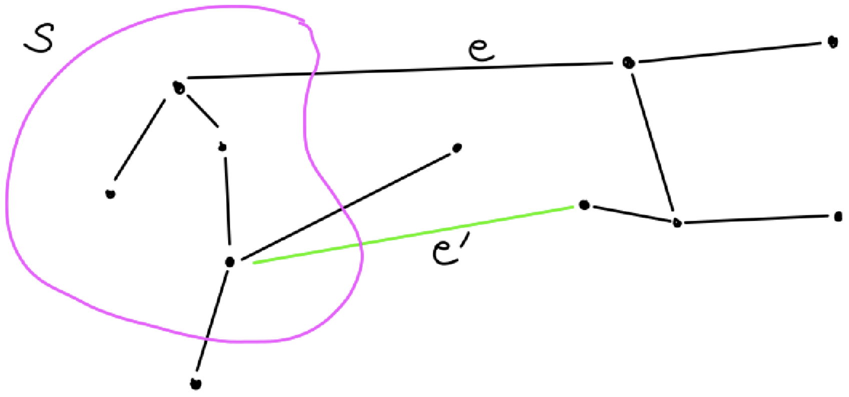
Proof of correctness of Kruskal's algorithm

Note: For this, we need to show that the graph G' that the algorithm derives is a minimum spanning tree.

Lemma (Cut Property) Presume that all edge weights in G are distinct. Let $e = (v, w)$ be the edge in G whose weight $l(e)$ is the smallest. Let $S \subset V$, $S \neq \emptyset$, be such that one endpoint of edge e is in S and the other endpoint is in $V|S$. Then every minimum spanning tree in G contains edge e.

Proof Suppose there is a minimum spanning tree (MST) T in G which does not contain edge e' (i.e. do a proof by contradiction), where e' denotes the edge in G with the smallest weight.

This implies that adding edge e' to T would yield a cycle C, i.e. T would no longer be a tree.



Let edge e be another edge of C that connects a node in S with one in $V|S$.
Let T' be the tree that we get from T by replacing e with e'.
Then

- T' is indeed a tree as it has $|V|$ nodes, has $|V|-1$ edges and it is connected
- T' is a spanning tree as it contains $|V|$ nodes
- $cost(T') < cost(T)$ because $l(e') < l(e)$ as e' is the edge in G with smallest weight
  - T' would be a MST rather than T
  - (Contradiction)
  - T is a MST which contains e'. □

Theorem Kruskal's algorithm retrieves a minimum spanning tree of G.

Observations: • K's algorithm explicitly avoids creating a subgraph with cycles.

Proof:

1. Let $e = (v, w)$ denote the edge that is about to be added by Kruskal's Algorithm and let S denote the set of all nodes to which v has a path (before edge e is added).
   Then $v \in S$, but $w \notin S$ as adding e according to Kruskal's algorithm does not create a cycle.
   As e is the edge to be added next, it is the cheapest edge of all edges linking S and $V|S$. According to the previous lemma, we know that edge e therefore belongs to every minimum spanning tree.
2. What we need to show to conclude the proof of the theorem is that Kruskal's algorithm returns a spanning tree of G.
   Let G' denote the subgraph returned by Kruskal's algorithm. We already know that G' contains no cycles as the algorithm explicitly avoids creating those.
3. We also know that G' is connected as we would otherwise have a non-empty subset S of nodes, $S \subset V$, with no edges connecting S and $V|S$. As the input graph G is known to be connected, there is always a connecting edge between any non-empty subset $S \subset V$ and $V|S$

From 2 and 3, G' is a spanning tree of G.

Then from 1, G' is a minimum spanning tree of G. □

Time Complexity of Kruskal's Algorithm (first thoughts):

Reminder: Kruskal's Algorithm

- start with a subgraph G' which contains only the isolated nodes from G (i.e. no edges)

(*) while the number of edges in G' is $< |V| - 1$ {

(*) of all the edges in G that have not yet been added to G', pick the edge e with the lowest $l(e)$

(*) add this edge to G' unless it creates a cycle in G' }

Assume Let graph G consist of $|V| = n$ nodes and $|E| = m$ edges

Observations

- as every execution of the while loop (*) picks exactly one edge, the while-loop is called m times
  - inside the while-loop, we have to set all edges in G that we have not yet been added to G' according to their cost (*) in order to identify the one with the lowest weight, and
  - we need to check (*) that this edge would not create a cycle in G'

Conclusion & Motivation for next section

- While we cannot lower the number of times m that the while-loop executes (*), we can probably lower the worst-case running time of tasks (*) and (*) by choosing clever data structures.

  Think about clever data structures to to use for implementing Kruskal's algorithm.