Greedy algorithms

October 20, 2017

Overview

1. Huffman codes
   - Existence of prefix codes
   - Shannon's theorem
   - Huffman algorithm

2. Minimal spanning trees
   - Graphs (some concepts and definitions)
   - Minimal spanning trees
   - Kruskal algorithm for MST
   - Prim's algorithm

Encoding a text file

- The problem: Suppose we have a text file made out of some finite alphabet $x_1, ..., x_n$ and we want to encode it to a binary code.
- It should look like:

$$\text{abcdefgh} \longrightarrow 110110001010101110110011010101$$

- We would like to encode the file such that:
  1. It is possible to reconstruct the original file from the encoded file (to decode the file).
  2. The encoded file is of minimal length.

Encoding a text file

- The problem: Suppose we have a text file made out of some finite alphabet $x_1, \ldots, x_n$ and we want to encode it to a binary code.
- It should look like:

$$\text{abcdefgh} \longrightarrow 110110001010101110110011010101$$

- We would like to encode the file such that:
  1. It is possible to reconstruct the original file from the encoded file (to decode the file).
  2. The encoded file is of minimal length.

Encoding a text file

- The problem: Suppose we have a text file made out of some finite alphabet $x_1, \ldots, x_n$ and we want to encode it to a binary code.
- It should look like:

$$\text{abcdefgh} \longrightarrow 1101100010101011101100110101 01$$

- We would like to encode the file such that:
  1. It is possible to reconstruct the original file from the encoded file (to decode the file).
  2. The encoded file is of minimal length.

Encoding a text file - an example

- Suppose the file has N characters all of them from the set $\{A, B, C, D\}$ and we know that the frequencies in which they appear in the file are:

character	A	B	C	D
frequency	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

- One possible encoding is:

character	A	B	C	D
Codeword	00	01	10	11

  In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 2 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 2 + \frac{N}{8} \cdot 2 = 2N \text{ bits}$$

- Another option is:

character	A	B	C	D
Codeword	0	10	110	111

  In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 1 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 3 + \frac{N}{8} \cdot 3 = 1.75N \text{ bits}$$

Which is better!

Encoding a text file - an example

- Suppose the file has N characters all of them from the set $\{A, B, C, D\}$ and we know that the frequencies in which they appear in the file are:

character	A	B	C	D
frequency	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

- One possible encoding is:

character	A	B	C	D
Codeword	00	01	10	11

In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 2 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 2 + \frac{N}{8} \cdot 2 = 2N \text{ bits}$$

- Another option is:

character	A	B	C	D
Codeword	0	10	110	111

In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 1 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 3 + \frac{N}{8} \cdot 3 = 1.75N \text{ bits}$$

Which is better!

Encoding a text file - an example

- Suppose the file has N characters all of them from the set $\{A, B, C, D\}$ and we know that the frequencies in which they appear in the file are:

character	A	B	C	D
frequency	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

- One possible encoding is:

character	A	B	C	D
Codeword	00	01	10	11

In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 2 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 2 + \frac{N}{8} \cdot 2 = 2N \text{ bits}$$

- Another option is:

character	A	B	C	D
Codeword	0	10	110	111

In this case, the length of the encoded file will be

$$\frac{N}{2} \cdot 1 + \frac{N}{4} \cdot 2 + \frac{N}{8} \cdot 3 + \frac{N}{8} \cdot 3 = 1.75N \text{ bits}$$

Which is better!

Prefix codes

- One way to make sure that we can decode the encoded file is to make sure that no code word is the prefix (beginning) of another codeword.
- Definition: A prefix code is a collection of binary codewords $w_1, w_2, \ldots, w_n$ such that for any $i \neq j$ the codeword w_i is not the prefix of w_j.
- Prefix codes makes the decoding process especially simple. We will only study prefix codes. However this is not the only possible way to go.

Prefix codes

- One way to make sure that we can decode the encoded file is to make sure that no code word is the prefix (beginning) of another codeword.
- Definition: A prefix code is a collection of binary codewords $w_1, w_2, ..., w_n$ such that for any $i \neq j$ the codeword w_i is not the prefix of w_j.
- Prefix codes makes the decoding process especially simple. We will only study prefix codes. However this is not the only possible way to go.

Prefix codes

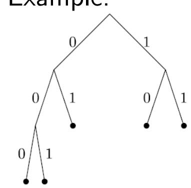
- One way to make sure that we can decode the encoded file is to make sure that no code word is the prefix (beginning) of another codeword.
- Definition: A prefix code is a collection of binary codewords $w_1, w_2, \ldots, w_n$ such that for any $i \neq j$ the codeword w_i is not the prefix of w_j.
- Prefix codes makes the decoding process especially simple. We will only study prefix codes. However this is not the only possible way to go.

Prefix codes and binary trees

- There is a one to one correspondence between prefix codes with m codewords and binary trees with m leaves.
- Example:
- We tag the edges going left by 0 and the edges going right by 1. Each leaf of the tree corresponds to a codeword according to the path from the root to the leaf.
- The collection of codewords that corresponds to the leaves of a binary tree is a prefix code. Moreover, any prefix code corresponds to some binary tree.

Prefix codes and binary trees

- There is a one to one correspondence between prefix codes with m codewords and binary trees with m leaves.
- Example:

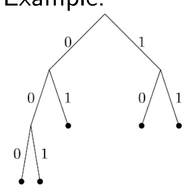


$\longleftrightarrow \quad \{000, 001, 01, 10, 11\}$

- We tag the edges going left by 0 and the edges going right by 1. Each leaf of the tree corresponds to a codeword according to the path from the root to the leaf.
- The collection of codewords that corresponds to the leaves of a binary tree is a prefix code. Moreover, any prefix code corresponds to some binary tree.

Prefix codes and binary trees

- There is a one to one correspondence between prefix codes with m codewords and binary trees with m leaves.
- Example:

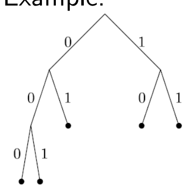


$\longleftrightarrow$ $\{000, 001, 01, 10, 11\}$

- We tag the edges going left by 0 and the edges going right by 1. Each leaf of the tree corresponds to a codeword according to the path from the root to the leaf.
- The collection of codewords that corresponds to the leaves of a binary tree is a prefix code. Moreover, any prefix code corresponds to some binary tree.

Prefix codes and binary trees

- There is a one to one correspondence between prefix codes with *m* codewords and binary trees with *m* leaves.
- Example:



$\longleftrightarrow \quad \{000, 001, 01, 10, 11\}$

- We tag the edges going left by 0 and the edges going right by 1. Each leaf of the tree corresponds to a codeword according to the path from the root to the leaf.
- The collection of codewords that corresponds to the leaves of a binary tree is a prefix code. Moreover, any prefix code corresponds to some binary tree.

Existence of prefix codes

- Clearly we can not find a prefix code for A,B and C with all lengths being 1.
  If A=0 and B=1 we have no codeword for C
- We can ask the following question:
  Given the lengthes $\{l_1, ..., l_n\}$,
  when can we find a prefix code $\{w_1, ..., w_n\}$ such that $|w_i| = l_i$?
- Any guesses ?
- The answer is on the next slide.

Existence of prefix codes

- Clearly we can not find a prefix code for A,B and C with all lengths being 1.
  If A=0 and B=1 we have no codeword for C
- We can ask the following question:
  Given the lengthes $\{l_1, ..., l_n\}$,
  when can we find a prefix code $\{w_1, ..., w_n\}$ such that $|w_i| = l_i$?
- Any guesses ?
- The answer is on the next slide.

Existence of prefix codes

- Clearly we can not find a prefix code for A,B and C with all lengths being 1.
  If A=0 and B=1 we have no codeword for C
- We can ask the following question:
  Given the lengthes $\{l_1, ..., l_n\}$,
  when can we find a prefix code $\{w_1, ..., w_n\}$ such that $|w_i| = l_i$?
- Any guesses ?
- The answer is on the next slide.

Existence of prefix codes

- Clearly we can not find a prefix code for A,B and C with all lengths being 1.
  If A=0 and B=1 we have no codeword for C
- We can ask the following question:
  Given the lengthes $\{l_1, ..., l_n\}$,
  when can we find a prefix code $\{w_1, ..., w_n\}$ such that $|w_i| = l_i$?
- Any guesses ?
- The answer is on the next slide.

Existence of prefix codes

- Theorem: There exists a prefix code $\{w_1, ..., w_n\}$ with lengths $l_1, ..., l_n$ if and only if:

$$\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$$

- Proof: By what we said about binary trees and prefix codes, it is enough to show that there exists a binary tree T with leaves $v_1, ..., v_n$ of heights $h_T(v_i) = l_i$ if and only if $\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$.
- Since the theorem is "if and only if" we will prove both directions.

Existence of prefix codes

- Theorem: There exists a prefix code $\{w_1, ..., w_n\}$ with lengths $l_1, ..., l_n$ if and only if:

$$\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$$

- Proof: By what we said about binary trees and prefix codes, it is enough to show that there exists a binary tree T with leaves $v_1, ..., v_n$ of heights $h_T(v_i) = l_i$ if and only if $\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$.
- Since the theorem is "if and only if" we will prove both directions.

Existence of prefix codes

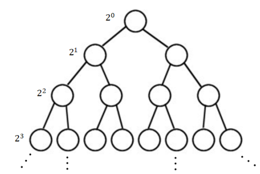
- Theorem: There exists a prefix code $\{w_1, ..., w_n\}$ with lengths $l_1, ..., l_n$ if and only if:

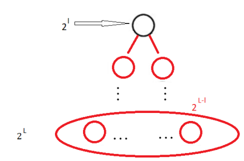
$$\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$$

- Proof: By what we said about binary trees and prefix codes, it is enough to show that there exists a binary tree T with leaves $v_1, ..., v_n$ of heights $h_T(v_i) = l_i$ if and only if $\sum_{i=1}^{i=n} 2^{-l_i} \leq 1$.
- Since the theorem is "if and only if" we will prove both directions.

Proof - first direction

- Assume there exists a binary tree T with leaves $v_1, \ldots, v_n$ of heights $h_T(v_i) = l_i$. Denote $L = \max_{1 \leq i \leq n}\{l_i\}$.
  If we complete each leaf of T as a full binary tree up to level L, we get a sub graph of the full binary tree of hight L, whose leaves are all of height L.





Proof - first direction

- A leaf in level l_i has 2^{L-l_i} descendants (in level L).
  The sets of descendants of the different leaves are disjoint.
  The number of all the descendants of the original leaves is no more than 2^L.
  Thus, we get:

$$2^L \geq \sum_{i=1}^{n} 2^{L-l_i} = 2^L \cdot \sum_{i=1}^{n} 2^{-l_i}$$

This implies that $1 \geq \sum_{i=1}^{n} 2^{-l_i}$.

Proof - second direction

On the other direction, assume that $1 \geq \sum_{i=1}^{n} 2^{-l_i}$.
We prove by induction on n (the number of codewords) that there exists a binary tree with n leaves of heights $l_1, ..., l_n$.
Assume, without loss of generality that $l_n = max_i\{l_1, ..., l_n\}$.
Since $1 \geq \sum_{i=1}^{n-1} 2^{-l_i}$, it follows from the induction hypothesis that there exists a binary tree T' with leaves $v_1, ..., v_{n-1}$ such that $h_{T'}(v_i) = l_i$.
If, like before, we complete all the leaves to full binary trees, up to level $L = l_n$, than we get $2^{l_n - l_i}$ leaves for each leaf in the original tree. It follows that:

$$\sum_{i=1}^{n-1} 2^{l_n - l_i} = 2^{l_n} \cdot \sum_{i=1}^{n-1} 2^{-l_i} \leq 2^{l_n}(1 - 2^{-l_n}) = 2^{l_n} - 1$$

We see that at least one node in level l_n is not a descendant of $v_1, ..., v_{n-1}$. If we take T to be T' and that node (with all his ancestors) we get the desired tree.■

Shannon's theorem

- Returning to the problem of finding an optimal prefix code: Given frequencies $p = (p_1, p_2, ..., p_n)$ such that $\sum_i p_i = 1$, we want to find a prefix code $w_1, ..., w_n$ with minimal:

$$\sum_{i=1}^{n} p_i |w_i|$$

- We denote the minimal value of $\sum_{i=1}^{n} p_i |w_i|$ by $f(p)$.
- We define the entropy function to be:

$$H(p_1, ..., p_n) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

- Shannon's theorem: We have:

$$H(p) \leq f(p) < H(p) + 1$$

Shannon's theorem

- Returning to the problem of finding an optimal prefix code: Given frequencies $p = (p_1, p_2, ..., p_n)$ such that $\sum_i p_i = 1$, we want to find a prefix code $w_1, ..., w_n$ with minimal:

$$\sum_{i=1}^{n} p_i |w_i|$$

- We denote the minimal value of $\sum_{i=1}^{n} p_i |w_i|$ by $f(p)$.
- We define the entropy function to be:

$$H(p_1, ..., p_n) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

- Shannon's theorem: We have:

$$H(p) \leq f(p) < H(p) + 1$$

Shannon's theorem

- Returning to the problem of finding an optimal prefix code: Given frequencies $p = (p_1, p_2, ..., p_n)$ such that $\sum_i p_i = 1$, we want to find a prefix code $w_1, ..., w_n$ with minimal:

$$\sum_{i=1}^{n} p_i |w_i|$$

- We denote the minimal value of $\sum_{i=1}^{n} p_i |w_i|$ by $f(p)$.
- We define the entropy function to be:

$$H(p_1, ..., p_n) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

- Shannon's theorem: We have:

$$H(p) \leq f(p) < H(p) + 1$$

Shannon's theorem

- Returning to the problem of finding an optimal prefix code: Given frequencies $p = (p_1, p_2, ..., p_n)$ such that $\sum_i p_i = 1$, we want to find a prefix code $w_1, ..., w_n$ with minimal:

$$\sum_{i=1}^{n} p_i |w_i|$$

- We denote the minimal value of $\sum_{i=1}^{n} p_i |w_i|$ by $f(p)$.
- We define the entropy function to be:

$$H(p_1, ..., p_n) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

- Shannon's theorem: We have:

$$H(p) \leq f(p) < H(p) + 1$$

Shannon's theorem - example

- For $p = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ we found a prefix code with $\sum_{i=1}^{n} p_i |w_i| = 1.75$.
  It follows that $f(p) \leq 1.75$.
  On the other hand:

$$H(p) = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{1}{8}\log_2 8 + \frac{1}{8}\log_2 8 =$$

$$= 0.5 + 0.5 + \frac{3}{8} + \frac{3}{8} = 1.75$$

- It follows that the code we found is optimal $(1.75 = H(p) \leq f(p))$.

Shannon's theorem - example

- For $p = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$ we found a prefix code with $\sum_{i=1}^{n} p_i |w_i| = 1.75$.
  It follows that $f(p) \leq 1.75$.
  On the other hand:

$$H(p) = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{1}{8}\log_2 8 + \frac{1}{8}\log_2 8 =$$

$$= 0.5 + 0.5 + \frac{3}{8} + \frac{3}{8} = 1.75$$

- It follows that the code we found is optimal $(1.75 = H(p) \leq f(p))$.

Shannon's theorem - proof (we will skip the proof)

- Proof: Given $p = (p_1, \ldots, p_n)$ we define $l_i = \left\lceil \log_2 \frac{1}{p_i} \right\rceil$.
  We get that:

$$\sum_i 2^{-l_i} \leq \sum_i 2^{-\log_2 \frac{1}{p_i}} = \sum_i p_i = 1$$

It follows that there exists a binary tree T with leaves $v_1, \ldots, v_n$ such that $h(v_i) = l_i$.
Therefore:

$$\underbrace{f(p)}_{\text{optimal}} \leq \sum_i p_i \cdot l_i = \sum_{i=1}^{n} p_i \cdot \left\lceil \log_2 \frac{1}{p_i} \right\rceil \leq$$

$$\sum_{i=1}^{n} p_i \cdot \left(\log_2 \frac{1}{p_i} + 1 \right) = H(p) + 1$$

Shannon's theorem - proof (we will skip the proof)

- Let $w_1, \dots, w_n$ be a prefix code with lengths $l_1, \dots, l_n$. We know that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. Therefore:

$$0 \geq \log_2\left(\sum_{i=1}^{n} 2^{-l_i}\right) = \log_2\left(\sum_{i=1}^{n} p_i \frac{2^{-l_i}}{p_i}\right)$$

$$\underbrace{\geq}_{\text{by the convexity of the function } \log_2(x)} \sum_{i=1}^{n} p_i \log_2 \frac{2^{-l_i}}{p_i}$$

$$= \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} + \sum_{i=1}^{n} p_i \log_2 2^{-l_i} = H(p) - \sum_{i=1}^{n} p_i l_i$$

■

(Note: the convexity here is actually concavity)

Huffman coding

- So far we haven't said anything about how to find an optimal prefix code.
- The Huffman Algorithm:
  (for an optimal prefix code for $p = (p_1, \ldots, p_n)$)
  1. Find the 2 minimal elements of $p = (p_1, \ldots, p_n)$ (without loss of generality $p_1, \ldots, p_{n-2} \geq p_{n-1}, p_n$).
  2. Find (recursively) an optimal binary tree for $(p_1, \ldots, p_{n-2}, p_{n-1} + p_n)$.
  3. Split the node (leaf) of $p_{n-1} + p_n$ to 2 nodes.

Huffman coding

- So far we haven't said anything about how to find an optimal prefix code.
- The Huffman Algorithm:
  (for an optimal prefix code for $p = (p_1, ..., p_n)$)
  1. Find the 2 minimal elements of $p = (p_1, ..., p_n)$ (without loss of generality $p_1, ..., p_{n-2} \geq p_{n-1}, p_n$).
  2. Find (recursively) an optimal binary tree for $(p_1, ..., p_{n-2}, p_{n-1} + p_n)$.
  3. Split the node (leaf) of $p_{n-1} + p_n$ to 2 nodes.

Huffman coding

- So far we haven't said anything about how to find an optimal prefix code.
- The Huffman Algorithm:
  (for an optimal prefix code for $p = (p_1, ..., p_n)$)
  1. Find the 2 minimal elements of $p = (p_1, ..., p_n)$ (without loss of generality $p_1, ..., p_{n-2} \geq p_{n-1}, p_n$).
  2. Find (recursively) an optimal binary tree for $(p_1, ..., p_{n-2}, p_{n-1} + p_n)$.
  3. Split the node (leaf) of $p_{n-1} + p_n$ to 2 nodes.

Huffman coding

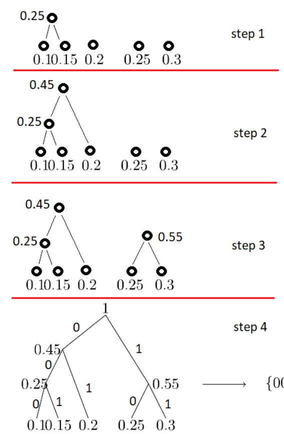
- So far we haven't said anything about how to find an optimal prefix code.
- The Huffman Algorithm:
  (for an optimal prefix code for $p = (p_1, ..., p_n)$)
  1. Find the 2 minimal elements of $p = (p_1, ..., p_n)$ (without loss of generality $p_1, ..., p_{n-2} \geq p_{n-1}, p_n$).
  2. Find (recursively) an optimal binary tree for $(p_1, ..., p_{n-2}, p_{n-1} + p_n)$.
  3. Split the node (leaf) of $p_{n-1} + p_n$ to 2 nodes.

Huffman coding

- So far we haven't said anything about how to find an optimal prefix code.
- The Huffman Algorithm:
  (for an optimal prefix code for $p = (p_1, ..., p_n)$)
  1. Find the 2 minimal elements of $p = (p_1, ..., p_n)$ (without loss of generality $p_1, ..., p_{n-2} \geq p_{n-1}, p_n$).
  2. Find (recursively) an optimal binary tree for $(p_1, ..., p_{n-2}, p_{n-1} + p_n)$.
  3. Split the node (leaf) of $p_{n-1} + p_n$ to 2 nodes.

Huffman coding - example

For $p = (0.1, 0.15, 0.2.0.25, 0.3)$



$\longrightarrow \{000, 001, 01, 10, 11\}$

Huffman coding

- The following theorem proves the correctness of the algorithm.
- Theorem: Let $p_1 \geq p_2 \geq ... \geq p_n$ be such that $\sum_{i=1}^{n} p_1 = 1$ $(p_i \geq 0)$ then:

$$f(p_1, p_2, ..., p_n) = f(p_1, p_2, ..., p_{n-1} + p_n) + p_{n-1} + p_n$$

(Remember that $f(p)$ is the value of $\sum_i p_i |w_i|$ where $(w_1, ..., w_n)$ is an optimal prefix code).

Huffman coding

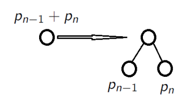
- The following theorem proves the correctness of the algorithm.
- Theorem: Let $p_1 \geq p_2 \geq ... \geq p_n$ be such that $\sum_{i=1}^{n} p_1 = 1$ $(p_i \geq 0)$ then:

$$f(p_1, p_2, ..., p_n) = f(p_1, p_2, ..., p_{n-1} + p_n) + p_{n-1} + p_n$$

(Remember that $f(p)$ is the value of $\sum_i p_i |w_i|$ where $(w_1, ..., w_n)$ is an optimal prefix code).

Huffman coding

proof: Let T' be an optimal tree for the probability vector $p' = (p_1, ..., p_{n-1} + p_n)$.
Let $v_1, ... v_{n-2}, v'_{n-1}$ be the leaves of T'. We build a new tree T by splitting the leaf v'_{n-1} into 2 nodes v_{n-1} and v_n.



The average length for the tree T that we get is:

$$\left(\sum_{i=1}^{n-2} p_i h_{T'}(v_i)\right) + p_{n-1}(h_{T'}(v'_{n-1}) + 1) + p_n(h_{T'}(v'_{n-1}) + 1) =$$

$$\left(\sum_{i=1}^{n-1} p_i h_{T'}(v_i)\right) + p_{n-1} + p_n$$

Huffman coding

proof: It follows that:

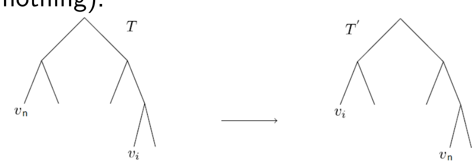
$$f(p_1, p_2, ..., p_n) \leq f(p_1, ..., p_{n-2}, p_{n-1} + p_n) + p_{n-1} + p_n$$

Huffman coding

proof: On the other hand, let T be an optimal tree for $p = (p_1, \ldots, p_n)$. If the leaves of T are $v_1, \ldots, v_n$ then we have:

$$f(p) = \sum_{i=1}^{i=n} p_i h_T(v_i)$$

Let i be such that the height of v_i is maximal and let T' be the tree that we get by exchanging v_i and v_n in T' (if $i = n$ we do nothing).



Huffman coding

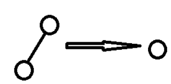
proof: T' must be an optimal tree for $(p_1, \dots, p_n)$ since:

$$\sum_{i=1}^{i=n} p_i h_T(v_i) - \sum_{i=1}^{i=n} p_i h_{T'}(v_i) =$$

$$(p_i h_T(v_i) + p_n h_T(v_n)) - (p_i h_T(v_n) + p_n h_T(v_i)) =$$

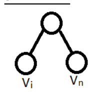
$$\underbrace{(p_i - p_n)}_{\geq 0} \underbrace{(h_T(v_i) - h_T(v_n))}_{\geq 0} \geq 0$$

It follows that there exists an optimal tree T such that $h_T(v_n)$ is maximal (among all the leaves).
Clearly v_n is not the only child of its parent in T, since then we could remove one edge and improve the optimality of T.



Huffman coding

proof: It follows that there are 2 children.



Since V_n has maximal height, it follows that the other child is also of maximal height.
Similar arguments show that we can make sure that the other child is v_{n-1}.
So far we saw that there exists an optimal T such that v_n and v_{n-1} have the same parent (and are of maximal height).
We consider a new tree T'' with leaves $u_1, ..., u_{n-2}, u_{n-1}$ where $u_i = v_i$ and $u_{n-1} = \text{parent}(v_n)$.

Huffman coding

proof: If we consider the frequencies vector $q = (q_1, \ldots, q_{n-1}) = (p_1, \ldots, p_{n-2}, p_{n-1} + p_n)$ then we see that:

$$f(q) \leq \sum_{i=1}^{n-1} h_{T''}(u_i) q_i = \sum_{i=1}^{n-2} h_T(v_i) p_i + (p_{n-1} + p_n)(\underbrace{h_T(v_n)}_{= h_T(v_{n-1})} - 1) =$$

$$= \sum_{i=1}^{n} h_T(v_i) p_i - (p_{n-1} + p_n) = f(p) - (p_{n-1} + p_n)$$

We see that:

$$f((p_1, \ldots, p_{n-2}, p_{n-1} + p_n)) = f(q) \leq f(p) - (p_{n-1} + p_n)$$

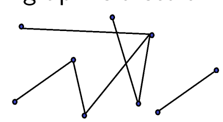
■

Huffman coding

The correctness of the Huffman algorithm follows.

Graphs

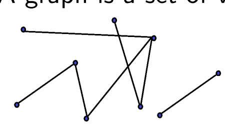
- A graph is a set of vertices and edges that connect them.



- More accurately, we have a set V of vertices (or nodes) and a set E of edges.
- An edge $e \in E$ is a pair of vertices $e = \{v, u\}$, $u, v \in V$.
- We denote $G = (V, E)$.
- A graph might be directed. In a directed graph the edges have a direction. The edges (u, v) and (v, u) are two different edges.

Graphs

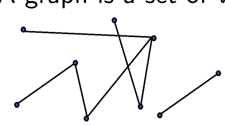
- A graph is a set of vertices and edges that connect them.



- More accurately, we have a set V of vertices (or nodes) and a set E of edges.
- An edge $e \in E$ is a pair of vertices $e = \{v, u\}$, $u, v \in V$.
- We denote $G = (V, E)$.
- A graph might be directed. In a directed graph the edges have a direction. The edges (u, v) and (v, u) are two different edges.

Graphs

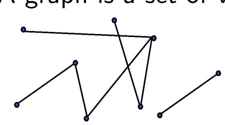
- A graph is a set of vertices and edges that connect them.



- More accurately, we have a set V of vertices (or nodes) and a set E of edges.
- An edge $e \in E$ is a pair of vertices $e = \{v, u\}$, $u, v \in V$.
- We denote $G = (V, E)$.
- A graph might be directed. In a directed graph the edges have a direction. The edges (u, v) and (v, u) are two different edges.

Graphs

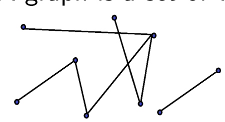
- A graph is a set of vertices and edges that connect them.



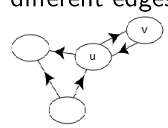
- More accurately, we have a set V of vertices (or nodes) and a set E of edges.
- An edge $e \in E$ is a pair of vertices $e = \{v, u\}$, $u, v \in V$.
- We denote $G = (V, E)$.
- A graph might be directed. In a directed graph the edges have a direction. The edges (u, v) and (v, u) are two different edges.

Graphs

- A graph is a set of vertices and edges that connect them.



- More accurately, we have a set V of vertices (or nodes) and a set E of edges.
- An edge $e \in E$ is a pair of vertices $e = \{v, u\}$, $u, v \in V$.
- We denote $G = (V, E)$.
- A graph might be directed. In a directed graph the edges have a direction. The edges (u, v) and (v, u) are two different edges.



Graphs

- We say that a graph $G = (V, E)$ is connected, if for any 2 vertices $u, v \in V$ there is a path from u to v:

$$u = v_0, v_1, ..., v_n = v \text{ such that}$$

$$(v_i, v_{i+1}) \in E \text{ for } 0 \leq i \leq n-1$$

- A simple cycle in a graph is a closed walk from one vertex to itself with no repetitions of vertices and edges except the first and last vertices:

$$u = v_0, v_1, ..., v_n = u \text{ s.t } (v_i, v_{i+1}) \in E \text{ and } v_i \neq v_j$$

$$\text{for any } i \neq j \text{ except } i = 0, j = n$$

Graphs

- We say that a graph $G = (V, E)$ is connected, if for any 2 vertices $u, v \in V$ there is a path from u to v:

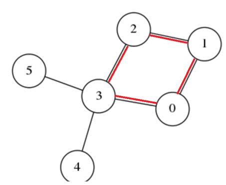
$$u = v_0, v_1, ..., v_n = v \text{ such that}$$

$$(v_i, v_{i+1}) \in E \text{ for } 0 \leq i \leq n - 1$$

- A simple cycle in a graph is a closed walk from one vertex to itself with no repetitions of vertices and edges except the first and last vertices:

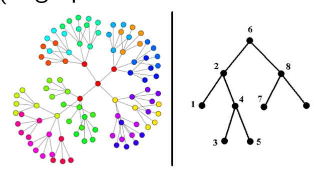
$$u = v_0, v_1, ..., v_n = u \text{ s.t } (v_i, v_{i+1}) \in E \text{ and } v_i \neq v_j$$

$$\text{for any } i \neq j \text{ except } i = 0, j = n$$



Graphs

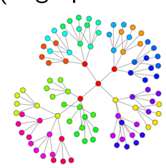
- A graph G is called a tree if it is connected and acyclic (has no simple cycles).
  (A graph G is called a forest if it is (only) acyclic)

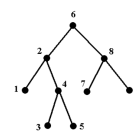


- We denote the number of connected components of a graph by $c(G)$.

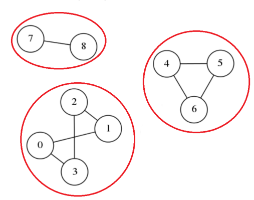
Graphs

- A graph G is called a tree if it is connected and acyclic (has no simple cycles).
  (A graph G is called a forest if it is (only) acyclic)



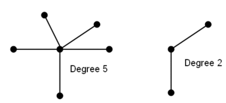


- We denote the number of connected components of a graph by $c(G)$.



Graphs

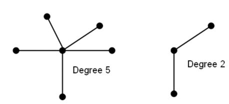
- The degree of a node v (denoted $deg(v)$) is the number of edges that has v as one of its vertices.



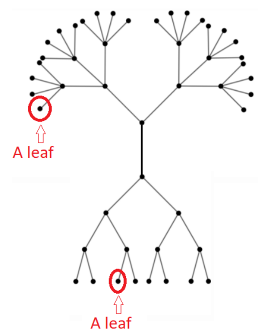
- A node v in a tree T is called a leaf if $deg(v) = 1$.

Graphs

- The degree of a node v (denoted $deg(v)$) is the number of edges that has v as one of its vertices.

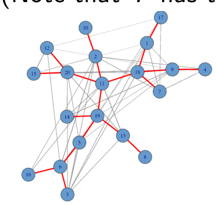


- A node v in a tree T is called a leaf if $deg(v) = 1$.



Graphs

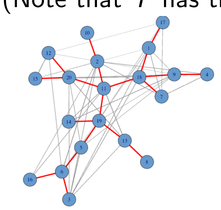
- It $G = (V, E)$ is a (connected) graph and $T = (V, E') \subseteq G$ is a tree than we say that T spans G (Note that T has the same set of vertices as G).



- Proposition:
  1. In any tree there exists a leaf.
  2. If $T = (V, E)$ is a tree then $|E| = |V| - 1$. More generally, for a forest G we have $|E| = |V| - c(G)$.
  3. Any connected graph $G = (V, E)$ has a spanning tree.

Graphs

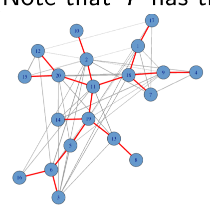
- It $G = (V, E)$ is a (connected) graph and $T = (V, E') \subseteq G$ is a tree than we say that T spans G (Note that T has the same set of vertices as G).



- Proposition:
  1. In any tree there exists a leaf.
  2. If $T = (V, E)$ is a tree then $|E| = |V| - 1$. More generally, for a forest G we have $|E| = |V| - c(G)$.
  3. Any connected graph $G = (V, E)$ has a spanning tree.

Graphs

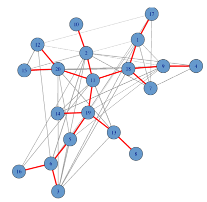
- It $G = (V, E)$ is a (connected) graph and $T = (V, E') \subseteq G$ is a tree than we say that T spans G (Note that T has the same set of vertices as G).



- Proposition:
  1. In any tree there exists a leaf.
  2. If $T = (V, E)$ is a tree then $|E| = |V| - 1$. More generally, for a forest G we have $|E| = |V| - c(G)$.
  3. Any connected graph $G = (V, E)$ has a spanning tree.

Graphs

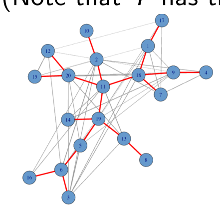
- It $G = (V, E)$ is a (connected) graph and $T = (V, E') \subseteq G$ is a tree than we say that T spans G (Note that T has the same set of vertices as G).



- Proposition:
  1. In any tree there exists a leaf.
  2. If $T = (V, E)$ is a tree then $|E| = |V| - 1$. More generally, for a forest G we have $|E| = |V| - c(G)$.
  3. Any connected graph $G = (V, E)$ has a spanning tree.

Graphs

- It $G = (V, E)$ is a (connected) graph and $T = (V, E') \subseteq G$ is a tree than we say that T spans G (Note that T has the same set of vertices as G).



- Proposition:
  1. In any tree there exists a leaf.
  2. If $T = (V, E)$ is a tree then $|E| = |V| - 1$. More generally, for a forest G we have $|E| = |V| - c(G)$.
  3. Any connected graph $G = (V, E)$ has a spanning tree.

Graphs

- Proof:
  1. Choose some node v_0. If v_0 is a leaf we're done. Otherwise we can move from v_0 to some v_1. If v_1 is a leaf we're done. Continuing like this, we never return to a node that we already visited since a tree has no cycles. It follows that (at some point) we must reach a leaf.
  2. We prove by induction on $|V|$. If $|V| = 1$ than clearly, $|E| = 0$ and $c(G) = 1$.
     Let $T \subseteq G$ be a connected component (T must be a tree). It follows from (1) that T has a leaf. Let v_0 be a leaf of T. If we remove v_0 from G we get a graph $G' = (V', E')$ with $|V'| = |V| - 1$, $|E'| = |E| - 1$ and $c(G') = c(G)$ (If T is just one vertex then $|V'| = |V| - 1$, $|E'| = |E|$ and $c(G') = c(G) - 1$). Therefore:

$$c(G) = c(G') = |V'| - |E'| = |V| - |E|$$

Graphs

- Proof:
  1. Choose some node v_0. If v_0 is a leaf we're done. Otherwise we can move from v_0 to some v_1. If v_1 is a leaf we're done. Continuing like this, we never return to a node that we already visited since a tree has no cycles. It follows that (at some point) we must reach a leaf.
  2. We prove by induction on $|V|$. If $|V| = 1$ than clearly, $|E| = 0$ and $c(G) = 1$.
     Let $T \subseteq G$ be a connected component (T must be a tree). It follows from (1) that T has a leaf. Let v_0 be a leaf of T. If we remove v_0 from G we get a graph $G' = (V', E')$ with $|V'| = |V| - 1$, $|E'| = |E| - 1$ and $c(G') = c(G)$ (If T is just one vertex then $|V'| = |V| - 1$, $|E'| = |E|$ and $c(G') = c(G) - 1$). Therefore:

$$c(G) = c(G') = |V'| - |E'| = |V| - |E|$$

Graphs

- Proof:
  1. Choose some node v_0. If v_0 is a leaf we're done. Otherwise we can move from v_0 to some v_1. If v_1 is a leaf we're done. Continuing like this, we never return to a node that we already visited since a tree has no cycles. It follows that (at some point) we must reach a leaf.
  2. We prove by induction on $|V|$. If $|V| = 1$ than clearly, $|E| = 0$ and $c(G) = 1$.
     Let $T \subseteq G$ be a connected component (T must be a tree). It follows from (1) that T has a leaf. Let v_0 be a leaf of T. If we remove v_0 from G we get a graph $G' = (V', E')$ with $|V'| = |V| - 1$, $|E'| = |E| - 1$ and $c(G') = c(G)$ (If T is just one vertex then $|V'| = |V| - 1$, $|E'| = |E|$ and $c(G') = c(G) - 1$). Therefore:

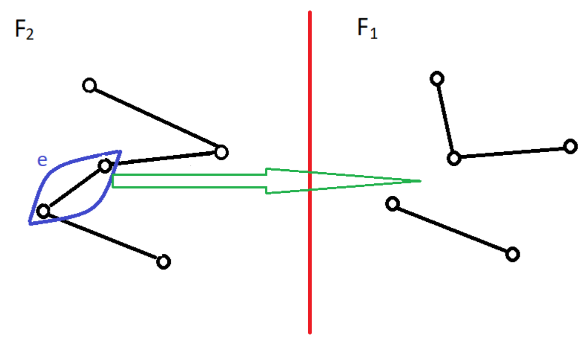
$$c(G) = c(G') = |V'| - |E'| = |V| - |E|$$

Graphs

- Proof: (3) By induction. Choose $v_0 \in V$ and remove it from G. By the induction hypothesis each connected component in $G - \{v_0\}$ has a minimal spanning tree. Let $T_1, \ldots, T_m$ be the spanning trees for the connected components of $G' = G - \{v_0\}$. Since G is connected, there exists an edge from (at least) one of the nodes in each of the trees $T1, \ldots, T_m$ to v_0. Adding one such edge from each T_i to v_0 we get a spanning tree T for G.

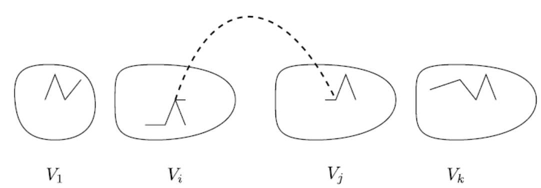
Graphs

- The following lemma is important:
  Let $F_1 = (V, E_1)$ and $F_2 = (V, E_2)$ be 2 forests on the same set of vertices. If $|E_2| > |E_1|$ then there exists an edge $e \in E_2 - E_1$ such that $(V, E_1 \cup \{e\})$ is a forest.



Graphs

- Proof: Let $k = c(F_1)$ and let $V_1, \ldots, V_k$ be the sets of vertices of the different connected component.
  For any i, the graph $(V_i, \binom{V_i}{2} \cap E_2)$ is a forest (since it has no cycles). It follows that $\left|\binom{V_i}{2} \cap E_2\right| \leq |V_i| - 1$ (otherwise it would have a cycle).
  If there exist $1 \leq i < j \leq k$ and an edge $(u, v) \in E_2$ such that $u \in V_i$ and $v \in V_j$ than $(V, E_1 \cup \{(u, v)\})$ is a forest.



Otherwise $E_2 \subseteq \bigcup_{i=1}^{k}(E_2 \cap \binom{V_i}{2})$. It follows that:

Graphs

- Proof:

$$|E_2| \leq \sum_{i=1}^{k} \left| E_2 \cap \binom{V_i}{2} \right| \leq \sum_{i=1}^{k} (|V_i| - 1) = |V| - k = |E_1|$$

But this is a contradiction. ■

Minimal spanning trees

- The problem: We are given a connected graph $G = (V, E)$ with different weights on its edges. The weights are given by $w : E \to \mathbb{R}_{>0}$. We wish to find a spanning tree $T = (V, E')$ of minimal weight.
  A spanning tree T with minimal $\sum_{e \in E'} w(e)$ is called a Minimal Spanning Tree (MST).
- Note that a minimal spanning tree is not necessarily unique.

Minimal spanning trees

- The problem: We are given a connected graph $G = (V, E)$ with different weights on its edges. The weights are given by $w : E \to \mathbb{R}_{>0}$. We wish to find a spanning tree $T = (V, E')$ of minimal weight.
  A spanning tree T with minimal $\sum_{e \in E'} w(e)$ is called a Minimal Spanning Tree (MST).
- Note that a minimal spanning tree is not necessarily unique.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
  - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
  - Step: After choosing $e_1, ..., e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, ..., e_k, e_{k+1}\})$ is acyclic.
  - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, ..., f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq ... \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
    - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
    - Step: After choosing $e_1, \ldots, e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, \ldots, e_k, e_{k+1}\})$ is acyclic.
    - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, \ldots, f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq \ldots \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
  - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
  - Step: After choosing $e_1, \dots, e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, \dots, e_k, e_{k+1}\})$ is acyclic.
  - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, \dots, f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq \dots \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
  - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
  - Step: After choosing $e_1, \ldots, e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, \ldots, e_k, e_{k+1}\})$ is acyclic.
  - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, \ldots, f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq \ldots \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
  - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
  - Step: After choosing $e_1, \dots, e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, \dots, e_k, e_{k+1}\})$ is acyclic.
  - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, \dots, f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq \dots \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

Kruskal algorithm for MST

- Kruskal algorithm finds a minimal spanning tree for a connected graph $G = (V, E)$ (we denote $|V| = n$):
  - Initialize: find $e_1 \in E$ with minimal weight. $w(e_1) = \min_{e \in E}\{w(e)\}$.
  - Step: After choosing $e_1, ..., e_k$, we choose e_{k+1} to be an edge of minimal weight such that $(V, \{e_1, ..., e_k, e_{k+1}\})$ is acyclic.
  - Finish: Once we choose e_{n-1} we are done.
- We will prove: Let $T' = (V, F)$ be some spanning tree of G (not necessarily a minimal one). If we order the edges of T' as $(f_1, ..., f_{n-1})$ such that $w(f_1) \leq w(f_2) \leq ... \leq w(f_{n-1})$ then we have:

$$w(e_k) \leq w(f_k) \text{ for any } 1 \leq k \leq n-1$$

- It follows immediately from the above proposition that the algorithm actually finds a MST.

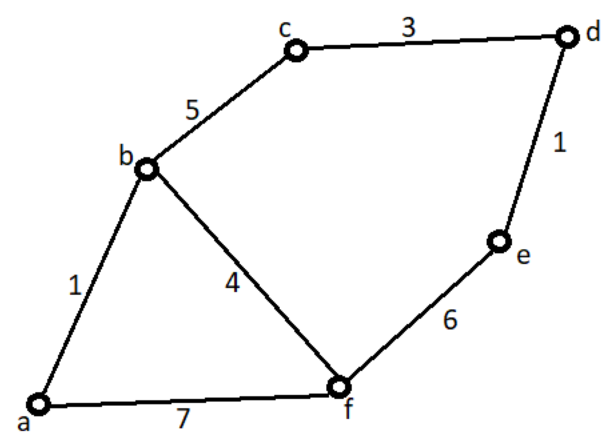
Kruskal algorithm for MST

- Proof: For f_1 and e_1 it follows from the definition of e_1.
  Denote $F_k = (f_1, ..., f_k)$ and $E_{k-1} = (e_1, ..., e_{k-1})$.
  The graphs (V, F_k) and (V, E_{k-1}) are both forests. Since $|F_k| > |E_{k-1}|$ it follows from the lemma we proved that there exists $f_i \in F_k$ such that $E_{k-1} \cup \{f_i\}$ is a forest.
  Therefore, by the way we choose e_k it follows that:

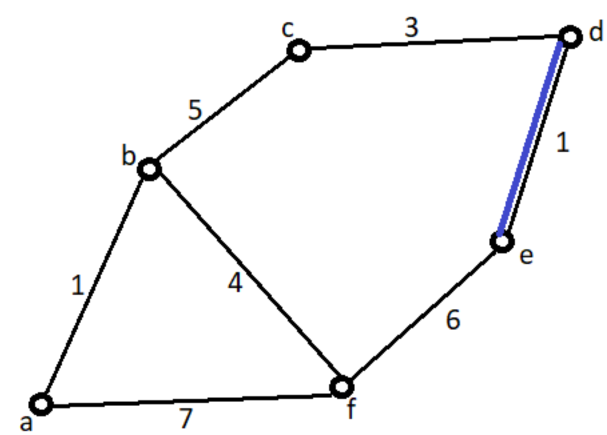
  $$w(e_k) \leq w(f_i) \leq w(f_k)$$

  ■

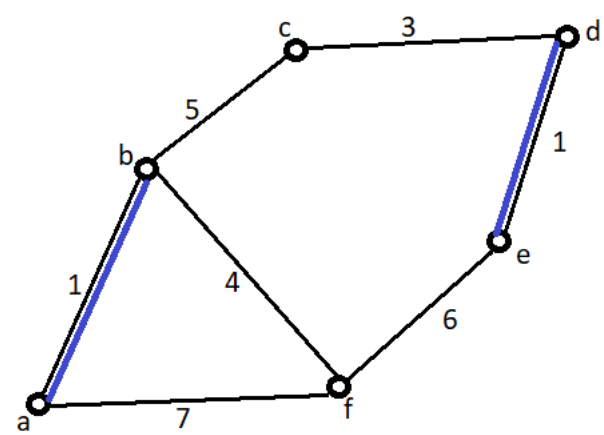
Kruskal algorithm for MST - example



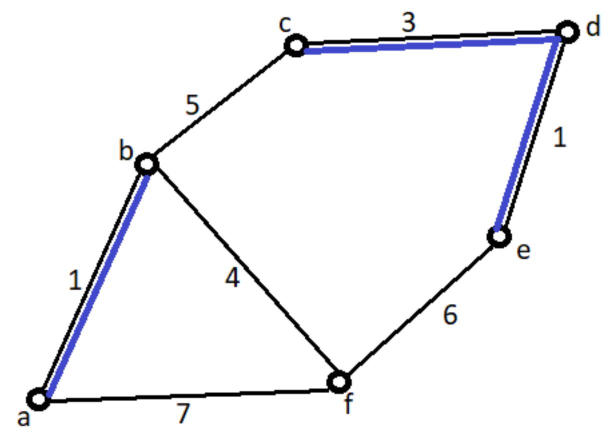
Kruskal algorithm for MST - example



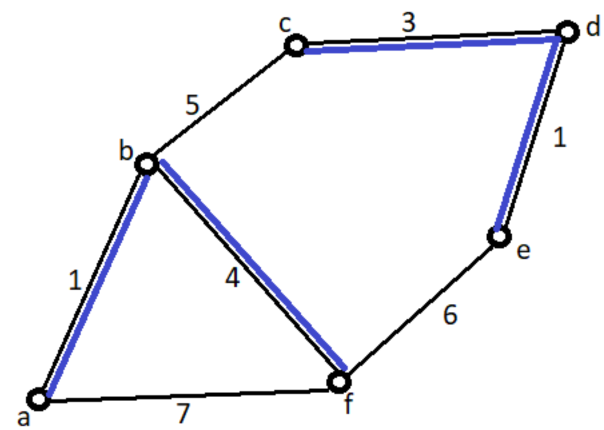
Kruskal algorithm for MST - example



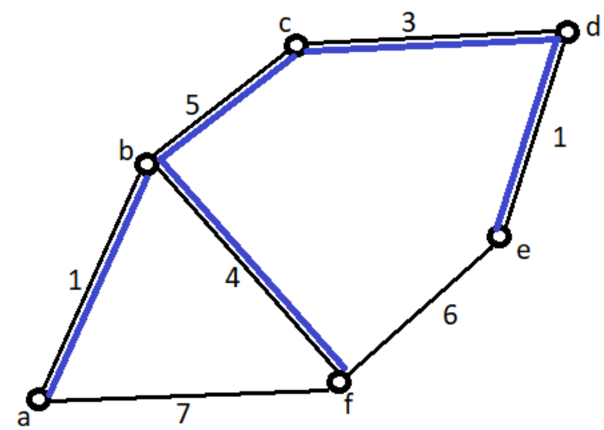
Kruskal algorithm for MST - example



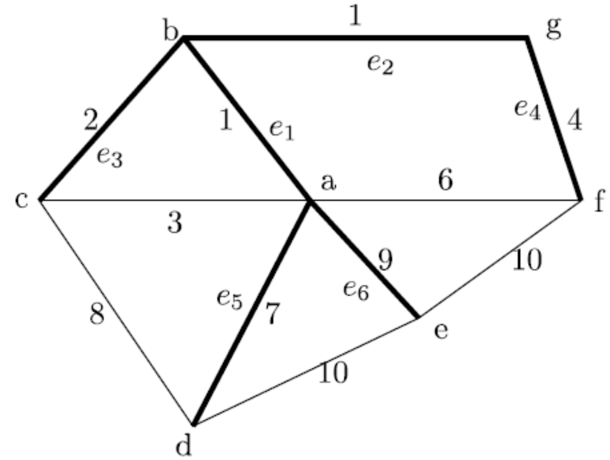
Kruskal algorithm for MST - example



Kruskal algorithm for MST - example



Kruskal algorithm for MST - another example



Kruskal algorithm for MST

- How do we implement the algorithm?
- Through the algorithm, we keep track of the connected components. At each step, we will denote the connected component of a vertex v by $A(v)$.
- We start by initializing $E' = \emptyset$ and $A(v) = \{v\}$ for any $v \in V$.
- We sort the edges by their weights: $w(e_1) \leq w(e_2) \leq ... \leq w(e_m)$.
- For $1 \leq k \leq m$: Let $e_k = (u, v)$. If $A(u) \neq A(v)$ then $E' = E' \cup \{e_k\}$ and:

$$A(x) = A(v) \cup A(v) \text{ for any } x \in A(u) \cup A(v)$$

Kruskal algorithm for MST

- How do we implement the algorithm?
- Through the algorithm, we keep track of the connected components. At each step, we will denote the connected component of a vertex v by $A(v)$.
- We start by initializing $E' = \emptyset$ and $A(v) = \{v\}$ for any $v \in V$.
- We sort the edges by their weights: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$.
- For $1 \leq k \leq m$: Let $e_k = (u, v)$. If $A(u) \neq A(v)$ then $E' = E' \cup \{e_k\}$ and:

$$A(x) = A(v) \cup A(v) \text{ for any } x \in A(u) \cup A(v)$$

Kruskal algorithm for MST

- How do we implement the algorithm?
- Through the algorithm, we keep track of the connected components. At each step, we will denote the connected component of a vertex v by $A(v)$.
- We start by initializing $E' = \emptyset$ and $A(v) = \{v\}$ for any $v \in V$.
- We sort the edges by their weights: $w(e_1) \leq w(e_2) \leq ... \leq w(e_m)$.
- For $1 \leq k \leq m$: Let $e_k = (u, v)$. If $A(u) \neq A(v)$ then $E' = E' \cup \{e_k\}$ and:

$$A(x) = A(v) \cup A(v) \text{ for any } x \in A(u) \cup A(v)$$

Kruskal algorithm for MST

- How do we implement the algorithm?
- Through the algorithm, we keep track of the connected components. At each step, we will denote the connected component of a vertex v by $A(v)$.
- We start by initializing $E' = \emptyset$ and $A(v) = \{v\}$ for any $v \in V$.
- We sort the edges by their weights: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$.
- For $1 \leq k \leq m$: Let $e_k = (u, v)$. If $A(u) \neq A(v)$ then $E' = E' \cup \{e_k\}$ and:

$$A(x) = A(v) \cup A(v) \text{ for any } x \in A(u) \cup A(v)$$

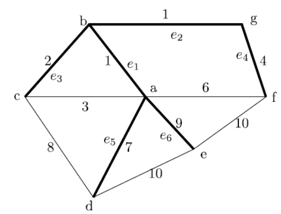
Kruskal algorithm for MST

- How do we implement the algorithm?
- Through the algorithm, we keep track of the connected components. At each step, we will denote the connected component of a vertex v by $A(v)$.
- We start by initializing $E' = \emptyset$ and $A(v) = \{v\}$ for any $v \in V$.
- We sort the edges by their weights: $w(e_1) \leq w(e_2) \leq ... \leq w(e_m)$.
- For $1 \leq k \leq m$: Let $e_k = (u, v)$. If $A(u) \neq A(v)$ then $E' = E' \cup \{e_k\}$ and:

$$A(x) = A(v) \cup A(v) \text{ for any } x \in A(u) \cup A(v)$$

Kruskal algorithm for MST - example

- For the example we saw we get:



E'	Connected components	Edges	Remark
Φ	a,b,c,d,e,f,g		
ab	ab,c,d,e,f,g	ab = 1	
ab,bg	abg,c,d,e,f	bg = 1	
ab,bg,bc	abgc,d,e,f	bc = 2	
		ac = 3	In the same component
ab,bg,bc,gf	abgcf,d,e	gf = 4	
		af = 6	In the same component
ab,bg,bc,gf,ad	abgcfd,e	ad = 7	
		cd = 8	In the same component
ab,bg,bc,gf,ad,ae	abgcfde	ae = 9	Here we can finish
		ef = 10	
		ed = 10	

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, \ldots, A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\to O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \to |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, ..., A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\to O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \to |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, ..., A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\rightarrow O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \rightarrow |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, ..., A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\to O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \to |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, ..., A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\rightarrow O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \rightarrow |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- At each step of the algorithm we hold n sets of connected components $A_1, ..., A_n$ (might be empty).
- In the beginning $|A_i| = 1$.
  In each step we have $t_i = |A_i|$ (this is just notation).
  For each $v \in V$ we maintain $i(v)$ such that $v \in A_{i(v)}$.
- In the k-th step we check the k-th smallest edge $e_k = (u, v)$:
  1. If $i(u) == i(v)$ - go to the next edge.
  2. If $i(u) \neq i(v)$ - If $t_{i(u)} > t_{i(v)}$ we move $A_{i(v)}$ into $A_{i(u)}$ and we update all of the relevant A's and $i(*)$'s.
- Counting operation:
  Sorting $\rightarrow O(|E| \log |E|)$.
  Comparing $i(u)$ and $i(v) \rightarrow |E|$.
  Step 2 is done for $|V| - 1$ edges. We will count how many operations are performed on the next slide.

Kruskal algorithm for MST - Complexity

- Claim: Each vertex is being moved / updated $O(\log|V|)$ times.
- Proof: Note that if we move $A_{i(u)}$ to $A_{i(v)}$ then we get a set at least twice the size of $A_{i(u)}$. It follows that a vertex can not move more than $\log_2|V|$ times.
- Even if for all the vertices we perform $\log|V|$ operations, we are still bounded by $O(|V|\log|V|)$.
- It follows that overall the complexity is:

$$O(|E|\log|E|) + O(|E|) + O(|V|\log|V|) = O(|E|\log|E|)$$

- Note that $|E| < |V|^2$ so $\log|E| < \log|V|^2 = 2\log|V|$. We see that:
$$O(|E|\log|E|) = O(|E|\log|V|)$$

Kruskal algorithm for MST - Complexity

- Claim: Each vertex is being moved / updated $O(\log|V|)$ times.
- Proof: Note that if we move $A_{i(u)}$ to $A_{i(v)}$ then we get a set at least twice the size of $A_{i(u)}$. It follows that a vertex can not move more than $\log_2|V|$ times.
- Even if for all the vertices we perform $\log|V|$ operations, we are still bounded by $O(|V|\log|V|)$.
- It follows that overall the complexity is:

$$O(|E|\log|E|) + O(|E|) + O(|V|\log|V|) = O(|E|\log|E|)$$

- Note that $|E| < |V|^2$ so $\log|E| < \log|V|^2 = 2\log|V|$. We see that:

$$O(|E|\log|E|) = O(|E|\log|V|)$$

Kruskal algorithm for MST - Complexity

- Claim: Each vertex is being moved / updated $O(\log|V|)$ times.
- Proof: Note that if we move $A_{i(u)}$ to $A_{i(v)}$ then we get a set at least twice the size of $A_{i(u)}$. It follows that a vertex can not move more than $\log_2|V|$ times.
- Even if for all the vertices we perform $\log|V|$ operations, we are still bounded by $O(|V|\log|V|)$.
- It follows that overall the complexity is:

$$O(|E|\log|E|) + O(|E|) + O(|V|\log|V|) = O(|E|\log|E|)$$

- Note that $|E| < |V|^2$ so $\log|E| < \log|V|^2 = 2\log|V|$. We see that:

$$O(|E|\log|E|) = O(|E|\log|V|)$$

Kruskal algorithm for MST - Complexity

- Claim: Each vertex is being moved / updated $O(\log|V|)$ times.
- Proof: Note that if we move $A_{i(u)}$ to $A_{i(v)}$ then we get a set at least twice the size of $A_{i(u)}$. It follows that a vertex can not move more than $\log_2|V|$ times.
- Even if for all the vertices we perform $\log|V|$ operations, we are still bounded by $O(|V|\log|V|)$.
- It follows that overall the complexity is:

$$O(|E|\log|E|) + O(|E|) + O(|V|\log|V|) = O(|E|\log|E|)$$

- Note that $|E| < |V|^2$ so $\log|E| < \log|V|^2 = 2\log|V|$. We see that:

$$O(|E|\log|E|) = O(|E|\log|V|)$$

Kruskal algorithm for MST - Complexity

- Claim: Each vertex is being moved / updated $O(\log|V|)$ times.
- Proof: Note that if we move $A_{i(u)}$ to $A_{i(v)}$ then we get a set at least twice the size of $A_{i(u)}$. It follows that a vertex can not move more than $\log_2|V|$ times.
- Even if for all the vertices we perform $\log|V|$ operations, we are still bounded by $O(|V|\log|V|)$.
- It follows that overall the complexity is:

$$O(|E|\log|E|) + O(|E|) + O(|V|\log|V|) = O(|E|\log|E|)$$

- Note that $|E| < |V|^2$ so $\log|E| < \log|V|^2 = 2\log|V|$. We see that:

$$O(|E|\log|E|) = O(|E|\log|V|)$$

Prim's algorithm

- We will now see Prim's algorithm for finding a MST for a graph $G = (V, E)$.
- The idea is:
  1. Choose a random node v_1 and define $V_1 = \{v_1\}$ and $E_1 = \emptyset$
  2. Define inductively a sequence of trees $T_k = (V_k, E_k)$ where $V_k = \{v_1, \ldots, v_k\} \subseteq V$ and $E_k = \{e_1, \ldots, e_{k-1}\} \subseteq E$ in the following way:
     Let $E(V_k, V - V_k)$ be all the edges $\{u, v\}$ of G such that $u \in V_k$ and $v \in V - V_k$ and let $e = \{u, v\} \in E(V_k, V - V_k)$ be an edge with minimal weight in $E(V_k, V - V_k)$.
     Define $e_k = e$, $v_{k+1} = v$ and $V_{k+1} = V_k \cup \{v_{k+1}\}$, $E_{k+1} = E_k \cup \{e_k\}$.
     Set $T_{k+1} = (V_{k+1}, E_{k+1})$.

Prim's algorithm

- We will now see Prim's algorithm for finding a MST for a graph $G = (V, E)$.
- The idea is:
  1. Choose a random node v_1 and define $V_1 = \{v_1\}$ and $E_1 = \emptyset$.
  2. Define inductively a sequence of **trees** $T_k = (V_k, E_k)$ where $V_k = \{v_1, \ldots, v_k\} \subseteq V$ and $E_k = \{e_1, \ldots, e_{k-1}\} \subseteq E$ in the following way:
     Let $E(V_k, V - V_k)$ be all the edges $\{u, v\}$ of G such that $u \in V_k$ and $v \in V - V_k$ and let $e = \{u, v\} \in E(V_k, V - V_k)$ be an edge with minimal weight in $E(V_k, V - V_k)$.
     Define $e_k = e$, $v_{k+1} = v$ and $V_{k+1} = V_k \cup \{v_{k+1}\}$, $E_{k+1} = E_k \cup \{e_k\}$.
     Set $T_{k+1} = (V_{k+1}, E_{k+1})$.

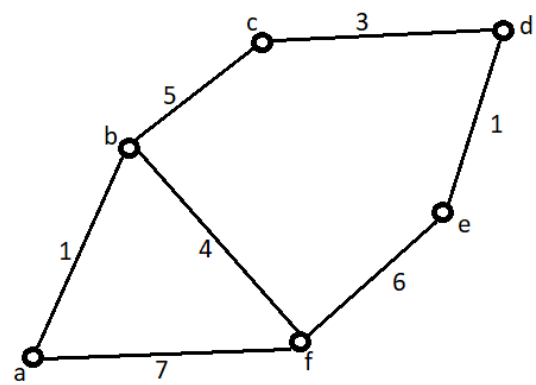
Prim's algorithm

- We will now see Prim's algorithm for finding a MST for a graph $G = (V, E)$.
- The idea is:
  1. Choose a random node v_1 and define $V_1 = \{v_1\}$ and $E_1 = \emptyset$.
  2. Define inductively a sequence of **trees** $T_k = (V_k, E_k)$ where $V_k = \{v_1, ..., v_k\} \subseteq V$ and $E_k = \{e_1, ..., e_{k-1}\} \subseteq E$ in the following way:
     Let $E(V_k, V - V_k)$ be all the edges $\{u, v\}$ of G such that $u \in V_k$ and $v \in V - V_k$ and let $e = \{u, v\} \in E(V_k, V - V_k)$ be an edge with minimal weight in $E(V_k, V - V_k)$.
     Define $e_k = e$, $v_{k+1} = v$ and $V_{k+1} = V_k \cup \{v_{k+1}\}$, $E_{k+1} = E_k \cup \{e_k\}$.
     Set $T_{k+1} = (V_{k+1}, E_{k+1})$.

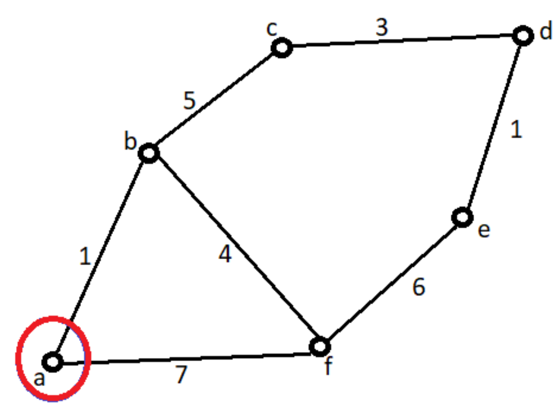
Prim's algorithm

- We will now see Prim's algorithm for finding a MST for a graph $G = (V, E)$.
- The idea is:
  1. Choose a random node v_1 and define $V_1 = \{v_1\}$ and $E_1 = \emptyset$.
  2. Define inductively a sequence of **trees** $T_k = (V_k, E_k)$ where $V_k = \{v_1, ..., v_k\} \subseteq V$ and $E_k = \{e_1, ..., e_{k-1}\} \subseteq E$ in the following way:
     Let $E(V_k, V - V_k)$ be all the edges $\{u, v\}$ of G such that $u \in V_k$ and $v \in V - V_k$ and let $e = \{u, v\} \in E(V_k, V - V_k)$ be an edge with minimal weight in $E(V_k, V - V_k)$.
     Define $e_k = e$, $v_{k+1} = v$ and $V_{k+1} = V_k \cup \{v_{k+1}\}$, $E_{k+1} = E_k \cup \{e_k\}$.
     Set $T_{k+1} = (V_{k+1}, E_{k+1})$.

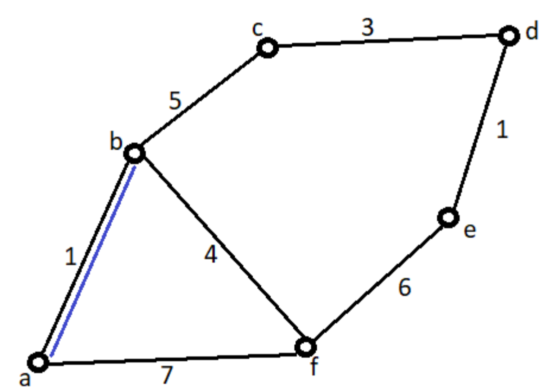
Prim's algorithm - example



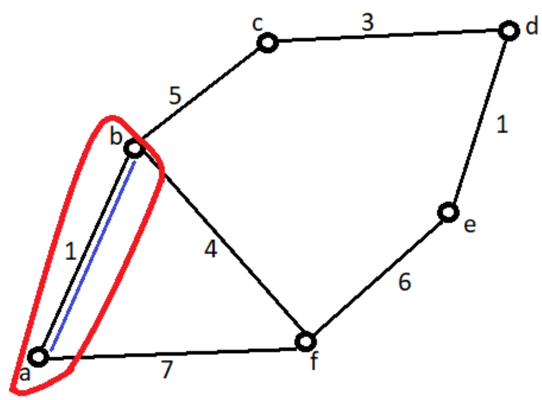
Prim's algorithm - example



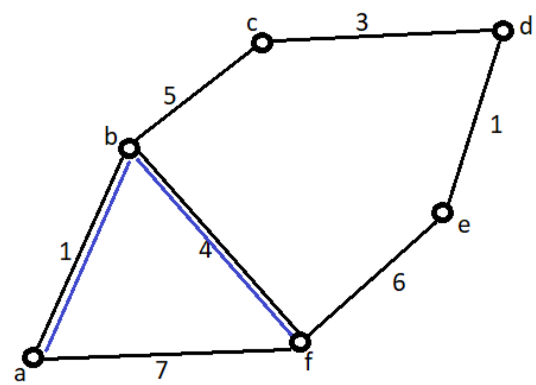
Prim's algorithm - example



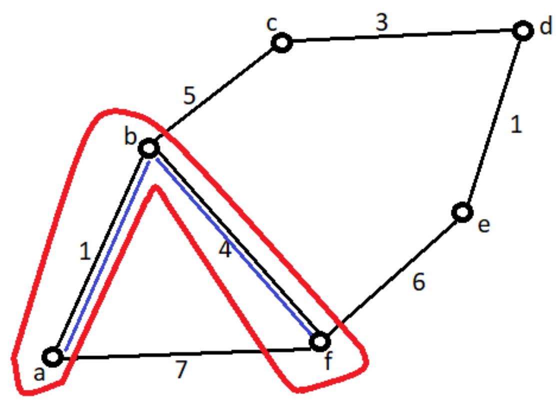
Prim's algorithm - example



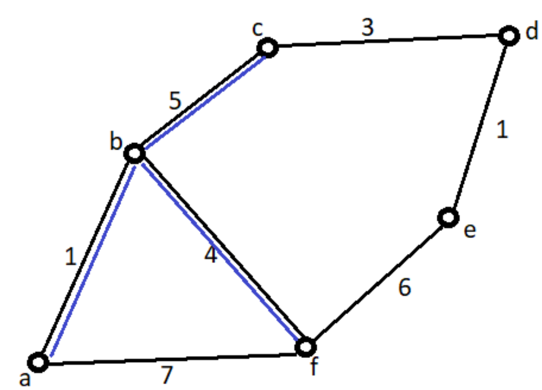
Prim's algorithm - example



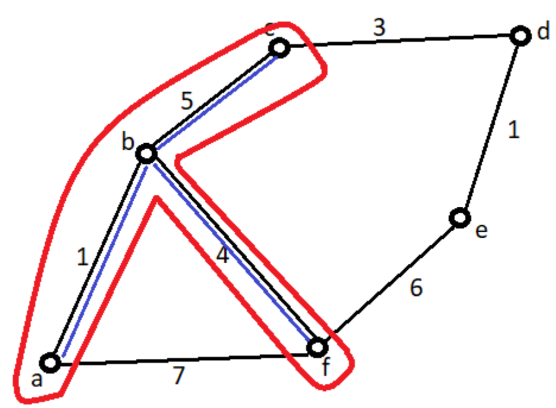
Prim's algorithm - example



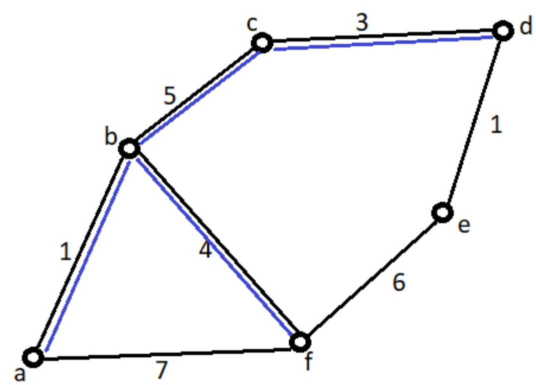
Prim's algorithm - example



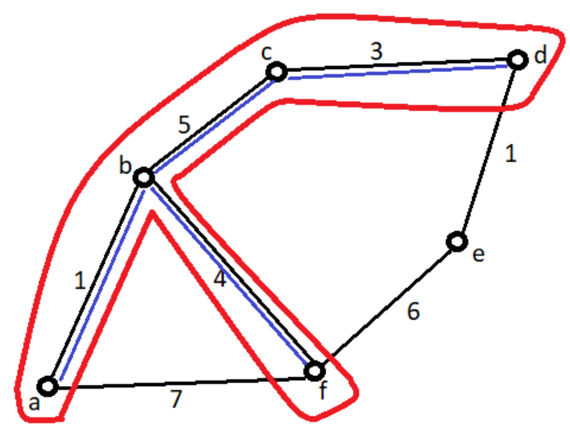
Prim's algorithm - example



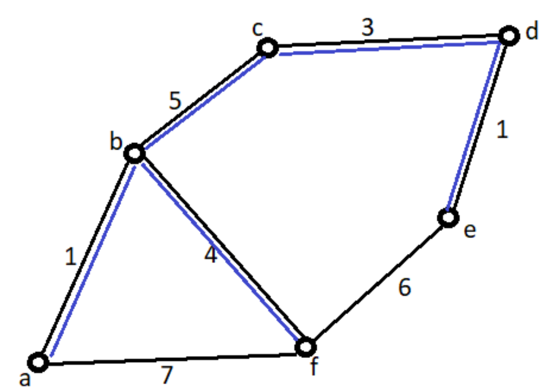
Prim's algorithm - example



Prim's algorithm - example



Prim's algorithm - example



Prim's algorithm - correctness

- Claim: For any $k \in \{1, 2, ..., n = |V|\}$, the tree T_k is contained in a Minimal Spanning Tree of G.
- The correctness of Prim's algorithm follows immediately since for $k = n$ we get that T_n is a MST.
- Proof: We prove by induction on k.
  For $k = 1$, there is only one vertex which is clearly in some MST.
  By our induction hypothesis there exists a MST $T = (V, F)$ such that $T_k \subseteq T$. If $e_k \in F$ than $T_{k+1} \subseteq T$ and we're done.

Prim's algorithm - correctness

- Claim: For any $k \in \{1, 2, ..., n = |V|\}$, the tree T_k is contained in a Minimal Spanning Tree of G.
- The correctness of Prim's algorithm follows immediately since for $k = n$ we get that T_n is a MST.
- Proof: We prove by induction on k.
  For $k = 1$, there is only one vertex which is clearly in some MST.
  By our induction hypothesis there exists a MST $T = (V, F)$ such that $T_k \subseteq T$. If $e_k \in F$ than $T_{k+1} \subseteq T$ and we're done.

Prim's algorithm - correctness

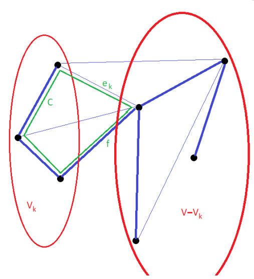
- Claim: For any $k \in \{1, 2, ..., n = |V|\}$, the tree T_k is contained in a Minimal Spanning Tree of G.
- The correctness of Prim's algorithm follows immediately since for $k = n$ we get that T_n is a MST.
- Proof: We prove by induction on k.
  For $k = 1$, there is only one vertex which is clearly in some MST.
  By our induction hypothesis there exists a MST $T = (V, F)$ such that $T_k \subseteq T$. If $e_k \in F$ than $T_{k+1} \subseteq T$ and we're done.

Prim's algorithm - correctness

Assume $e_k \notin F$. It follows that the graph with edges $F \cup \{e_k\}$ has a cycle C. Let $f \in (C - e_k)$ be such that $f \in E(V_k, V - V_k)$. From the definition of e_k it follows that $w(e_k) \leq w(f)$.
Set $T' = (V, (F - \{f\}) \cup \{e_k\})$. It follows that T' is a tree and:

$$w(T') = w(T) - w(f) + w(e_k) \leq w(T)$$

We see that T' is a MST and $T_k \subseteq T'$. ■



Prim's algorithm - implementation

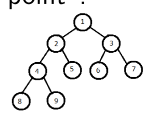
- We will require the notion of a heap.
- Informally: a binary tree is complete if all its levels are full except maybe the last one which is "full up to some point":
- If T is a complete binary tree such that each node v has a weight value $w(v)$, then we say that T is a heap if:

$$w(v) \leq w(\text{left}(v)) \text{ and } w(v) \leq w(\text{right}(v))$$

  whenever left(v) and right(v) are defined.
- For a node v we denote by T_v the sub-tree whose root is v.

Prim's algorithm - implementation

- We will require the notion of a heap.
- Informally: a binary tree is complete if all its levels are full except maybe the last one which is "full up to some point":



We can view an array as a complet tree.
This is an example for:
A=1,2,...,9
The indeices satisfies :
Left(v) = 2v , right(v) = 2v+1

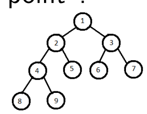
- If T is a complete binary tree such that each node v has a weight value $w(v)$, then we say that T is a heap if:

$$w(v) \leq w(\text{left}(v)) \text{ and } w(v) \leq w(\text{right}(v))$$

whenever left(v) and right(v) are defined.
- For a node v we denote by T_v the sub-tree whose root is v.

Prim's algorithm - implementation

- We will require the notion of a heap.
- Informally: a binary tree is complete if all its levels are full except maybe the last one which is "full up to some point":



We can view an array as a complet tree.
This is an example for:
A=1,2,...,9
The indeices satisfies:
Left(v) = 2v , right(v) = 2v+1

- If T is a complete binary tree such that each node v has a weight value $w(v)$, then we say that T is a heap if:

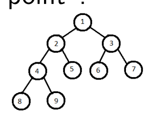
$$w(v) \leq w(\text{left}(v)) \text{ and } w(v) \leq w(\text{right}(v))$$

whenever left(v) and right(v) are defined.

- For a node v we denote by T_v the sub-tree whose root is v.

Prim's algorithm - implementation

- We will require the notion of a heap.
- Informally: a binary tree is complete if all its levels are full except maybe the last one which is "full up to some point":



We can view an array as a complet tree.
This is an example for:
A=1,2,...,9
The indeices satisfies:
Left(v) = 2v , right(v) = 2v+1

- If T is a complete binary tree such that each node v has a weight value $w(v)$, then we say that T is a heap if:

$$w(v) \leq w(\text{left}(v)) \text{ and } w(v) \leq w(\text{right}(v))$$

whenever left(v) and right(v) are defined.
- For a node v we denote by T_v the sub-tree whose root is v.

Prim's algorithm - implementation

- Assume $v \in V$ is a root of a complete tree such that $T_{\text{left}(v)}$ and $T_{\text{right}(v)}$ are heaps. The following algorithm turns T into a heap.
- Heapify(T)
  1. if $w(v) \leq \min(w(\text{left}(v)), w(\text{right}(v)))$ stop.
  2. else if $w(v) > w(\text{left}(v))$ and $w(\text{left}(v)) < w(\text{right}(v))$
  3. exchange $w(v)$ and $w(\text{left}(v))$
  4. Heapify($T_{\text{left}(v)}$)
  5. else
  6. exchange $w(v)$ and $w(\text{right}(v))$
  7. Heapify($T_{\text{right}(v)}$)
- Clearly the complexity of Heapify is $O(\text{height}(T))$.

Prim's algorithm - implementation

- Assume $v \in V$ is a root of a complete tree such that $T_{\text{left}(v)}$ and $T_{\text{right}(v)}$ are heaps. The following algorithm turns T into a heap.
- Heapify(T)
  1. If $w(v) \leq \min(w(\text{left}(v)), w(\text{right}(v)))$ stop.
  2. else if $w(v) > w(\text{left}(v))$ and $w(\text{left}(v)) < w(\text{right}(v))$
  3. exchange $w(v)$ and $w(\text{left}(v))$
  4. Heapify($T_{\text{left}(v)}$)
  5. else
  6. exchange $w(v)$ and $w(\text{right}(v))$
  7. Heapify($T_{\text{right}(v)}$)
- Clearly the complexity of Heapify is $O(\text{height}(T))$.

Prim's algorithm - implementation

- Assume $v \in V$ is a root of a complete tree such that $T_{\text{left}(v)}$ and $T_{\text{right}(v)}$ are heaps. The following algorithm turns T into a heap.
- Heapify(T)
  1. If $w(v) \leq \min(w(\text{left}(v)), w(\text{right}(v)))$ stop.
  2. else if $w(v) > w(\text{left}(v))$ and $w(\text{left}(v)) < w(\text{right}(v))$
  3. exchange $w(v)$ and $w(\text{left}(v))$
  4. Heapify($T_{\text{left}(v)}$)
  5. else
  6. exchange $w(v)$ and $w(\text{right}(v))$
  7. Heapify($T_{\text{right}(v)}$)
- Clearly the complexity of Heapify is $O(\text{height}(T))$.

Prim's algorithm - implementation

- If A is an array (of length n) considered as a complete tree T, we can build a heap out of it by:
  MakeHeap(A,n)
  1. For $i = n$ down to 1
  2. Heapify(T_i)
- Let m be such that $2^m \leq n \leq 2^{m+1} - 1$. For the complexity of MakeHeap we have:

$$\sum_{i=1}^{n} \text{height}(T_i) = O\left(\sum_{i=1}^{2^{m+1}-1} \text{height}(T_i)\right) =$$

$$O\left(\sum_{j=1}^{m} (m-j)2^j\right) = O\left(\sum_{j=1}^{m} j2^{m-j}\right) = O(2^m)\sum_{j=1}^{\infty} j2^{-j} = O(n)$$

Since $\sum_{j=1}^{\infty} j2^{-j} = 2$.

Prim's algorithm - implementation

- If A is an array (of length n) considered as a complete tree T, we can build a heap out of it by:
  MakeHeap(A,n)
  1. For $i = n$ down to 1
  2. Heapify(T_i)
- Let m be such that $2^m \leq n \leq 2^{m+1} - 1$. For the complexity of MakeHeap we have:

$$\sum_{i=1}^{n} \text{height}(T_i) = O\left(\sum_{i=1}^{2^{m+1}-1} \text{height}(T_i)\right) =$$

$$O\left(\sum_{j=1}^{m} (m-j)2^j\right) = O\left(\sum_{j=1}^{m} j2^{m-j}\right) = O(2^m)\sum_{j=1}^{\infty} j2^{-j} = O(n)$$

Since $\sum_{j=1}^{\infty} j2^{-j} = 2$.

Prim's algorithm - implementation

- If A is an array (of length n) considered as a complete tree T, we can build a heap out of it by:
  MakeHeap(A,n)
  1. For $i = n$ down to 1
  2. Heapify(T_i)
- Let m be such that $2^m \leq n \leq 2^{m+1} - 1$. For the complexity of MakeHeap we have:

$$\sum_{i=1}^{n} \text{height}(T_i) = O\left(\sum_{i=1}^{2^{m+1}-1} \text{height}(T_i)\right) =$$

$$O\left(\sum_{j=1}^{m}(m-j)2^j\right) = O\left(\sum_{j=1}^{m} j2^{m-j}\right) = O(2^m)\sum_{j=1}^{\infty} j2^{-j} = O(n)$$

Since $\sum_{j=1}^{\infty} j2^{-j} = 2$.

Prim's algorithm - implementation

- If A is an array (of length n) considered as a complete tree T, we can build a heap out of it by:
  MakeHeap(A,n)
  1. For $i = n$ down to 1
  2. Heapify(T_i)
- Let m be such that $2^m \leq n \leq 2^{m+1} - 1$. For the complexity of MakeHeap we have:

$$\sum_{i=1}^{n} \text{height}(T_i) = O\left(\sum_{i=1}^{2^{m+1}-1} \text{height}(T_i)\right) =$$

$$O\left(\sum_{j=1}^{m} (m-j)2^j\right) = O\left(\sum_{j=1}^{m} j2^{m-j}\right) = O(2^m)\sum_{j=1}^{\infty} j2^{-j} = O(n)$$

Since $\sum_{j=1}^{\infty} j2^{-j} = 2$.

Prim's algorithm - implementation

- Going back to Prim's algorithm.
- At each step of the algorithm we hold a tree $T_k = (V_k, E_k)$ ($V_k = \{v_1, ..., v_k\}$ and $E_k = \{e_1, ..., e_{k-1}\}$) and a heap whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of the edges in G.
  1. Initialization: $V_k = \{v_1\}$, $E_k = \emptyset$. H_1 a heap of the edges of the cut $(V, V - v_1)$
  2. Step: Take the edge $e = (u, v)$ from the root of H_k (where $u \in V_k$, $v \in V - V_k$). Define $e_k = e$, $v_{k+1} = v$ and update $V_{k+1} = V_k \cup \{v_{k+1}\}, E_{k+1} = E_k \cup \{e_k\}$. update H_k to H_{k+1}.
- How do we update H_k?

Prim's algorithm - implementation

- Going back to Prim's algorithm.
- At each step of the algorithm we hold a tree $T_k = (V_k, E_k)$ ($V_k = \{v_1, ..., v_k\}$ and $E_k = \{e_1, ..., e_{k-1}\}$) and a heap whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of the edges in G.
  1. Initialization: $V_k = \{v_1\}$, $E_k = \emptyset$. H_1 a heap of the edges of the cut $(V, V - v_1)$
  2. Step: Take the edge $e = (u, v)$ from the root of H_k (where $u \in V_k$, $v \in V - V_k$). Define $e_k = e$, $v_{k+1} = v$ and update $V_{k+1} = V_k \cup \{v_{k+1}\}, E_{k+1} = E_k \cup \{e_k\}$. update H_k to H_{k+1}.
- How do we update H_k?

Prim's algorithm - implementation

- Going back to Prim's algorithm.
- At each step of the algorithm we hold a tree $T_k = (V_k, E_k)$ ($V_k = \{v_1, ..., v_k\}$ and $E_k = \{e_1, ..., e_{k-1}\}$) and a heap whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of the edges in G.
  1. Initialization: $V_k = \{v_1\}$, $E_k = \emptyset$. H_1 a heap of the edges of the cut $(V, V - v_1)$
  2. Step: Take the edge $e = (u, v)$ from the root of H_k (where $u \in V_k$, $v \in V - V_k$). Define $e_k = e$, $v_{k+1} = v$ and update $V_{k+1} = V_k \cup \{v_{k+1}\}, E_{k+1} = E_k \cup \{e_k\}$. update H_k to H_{k+1}.
- How do we update H_k?

Prim's algorithm - implementation

- Going back to Prim's algorithm.
- At each step of the algorithm we hold a tree $T_k = (V_k, E_k)$ ($V_k = \{v_1, ..., v_k\}$ and $E_k = \{e_1, ..., e_{k-1}\}$) and a heap whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of the edges in G.
  1. Initialization: $V_k = \{v_1\}$, $E_k = \emptyset$. H_1 a heap of the edges of the cut $(V, V - v_1)$
  2. Step: Take the edge $e = (u, v)$ from the root of H_k (where $u \in V_k$, $v \in V - V_k$). Define $e_k = e$, $v_{k+1} = v$ and update $V_{k+1} = V_k \cup \{v_{k+1}\}, E_{k+1} = E_k \cup \{e_k\}$.
  update H_k to H_{k+1}.
- How do we update H_k?

Prim's algorithm - implementation

- Going back to Prim's algorithm.
- At each step of the algorithm we hold a tree $T_k = (V_k, E_k)$ ($V_k = \{v_1, ..., v_k\}$ and $E_k = \{e_1, ..., e_{k-1}\}$) and a heap whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of the edges in G.
  1. Initialization: $V_k = \{v_1\}$, $E_k = \emptyset$. H_1 a heap of the edges of the cut $(V, V - v_1)$
  2. Step: Take the edge $e = (u, v)$ from the root of H_k (where $u \in V_k$, $v \in V - V_k$). Define $e_k = e$, $v_{k+1} = v$ and update $V_{k+1} = V_k \cup \{v_{k+1}\}, E_{k+1} = E_k \cup \{e_k\}$.
     update H_k to H_{k+1}.
- How do we update H_k?

Prim's algorithm - implementation

- When updating H_k to H_{k+1} we drop all edges of the form (x, v_{k+1}) where $x \in V_k$ and we add all edges of the form (v_{k+1}, y) where $y \in V - V_{k+1}$.
- Removing an edge can be done using Heapify and thus in $O(\log |E|)$.
- Adding an edge can also be done in $O(\log |E|)$. (Simply add the edge as a leaf and "move it up level by level" if necessary.)
- At each step we do $\deg(v_{k+1})$ removals and additions of edges so overall we do $O(\deg(v_{k+1}) \log |E|)$ operations at each step.
- It follows that the overall complexity is:

$$O\left(\log |E| \sum_{v \in V} \deg(v)\right) = O((\log |E|) \cdot (2|E|)) = O(|E| \log |E|) = O(|E| \log |V|)$$

Prim's algorithm - implementation

- When updating H_k to H_{k+1} we drop all edges of the form (x, v_{k+1}) where $x \in V_k$ and we add all edges of the form (v_{k+1}, y) where $y \in V - V_{k+1}$.
- Removing an edge can be done using Heapify and thus in $O(\log |E|)$.
- Adding an edge can also be done in $O(\log |E|)$. (Simply add the edge as a leaf and "move it up level by level" if necessary.)
- At each step we do $\deg(v_{k+1})$ removals and additions of edges so overall we do $O(\deg(v_{k+1}) \log |E|)$ operations at each step.
- It follows that the overall complexity is:

$$O\left(\log |E| \sum_{v \in V} \deg(v)\right) = O((\log |E|) \cdot (2|E|)) = O(|E| \log |E|) = O(|E| \log |V|)$$

Prim's algorithm - implementation

- When updating H_k to H_{k+1} we drop all edges of the form (x, v_{k+1}) where $x \in V_k$ and we add all edges of the form (v_{k+1}, y) where $y \in V - V_{k+1}$.
- Removing an edge can be done using Heapify and thus in $O(\log |E|)$.
- Adding an edge can also be done in $O(\log |E|)$. (Simply add the edge as a leaf and "move it up level by level" if necessary.)
- At each step we do $\deg(v_{k+1})$ removals and additions of edges so overall we do $O(\deg(v_{k+1}) \log |E|)$ operations at each step.
- It follows that the overall complexity is:

$$O\left(\log |E| \sum_{v \in V} \deg(v)\right) = O((\log |E|) \cdot (2|E|)) = O(|E| \log |E|) = O(|E| \log |V|)$$

Prim's algorithm - implementation

- When updating H_k to H_{k+1} we drop all edges of the form (x, v_{k+1}) where $x \in V_k$ and we add all edges of the form (v_{k+1}, y) where $y \in V - V_{k+1}$.
- Removing an edge can be done using Heapify and thus in $O(\log |E|)$.
- Adding an edge can also be done in $O(\log |E|)$. (Simply add the edge as a leaf and "move it up level by level" if necessary.)
- At each step we do $\deg(v_{k+1})$ removals and additions of edges so overall we do $O(\deg(v_{k+1}) \log |E|)$ operations at each step.
- It follows that the overall complexity is:

$$O\left(\log |E| \sum_{v \in V} \deg(v)\right) = O((\log |E|) \cdot (2|E|)) = O(|E| \log |E|) = O(|E| \log |V|)$$

Prim's algorithm - implementation

- When updating H_k to H_{k+1} we drop all edges of the form (x, v_{k+1}) where $x \in V_k$ and we add all edges of the form (v_{k+1}, y) where $y \in V - V_{k+1}$.
- Removing an edge can be done using Heapify and thus in $O(\log |E|)$.
- Adding an edge can also be done in $O(\log |E|)$. (Simply add the edge as a leaf and "move it up level by level" if necessary.)
- At each step we do $\deg(v_{k+1})$ removals and additions of edges so overall we do $O(\deg(v_{k+1}) \log |E|)$ operations at each step.
- It follows that the overall complexity is:

$$O\left(\log |E| \sum_{v \in V} \deg(v)\right) = O((\log |E|) \cdot (2|E|)) = O(|E| \log |E|) = O(|E| \log |V|)$$