TTIC 31010, CMSC 37000-1

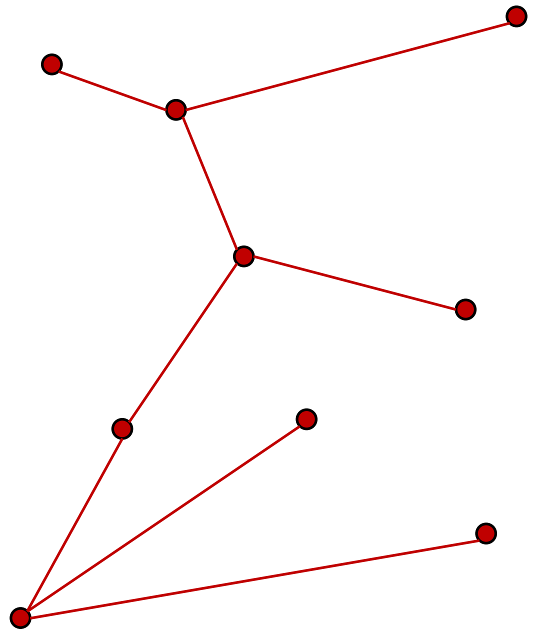
Lecture 2: MST and Huffman Coding

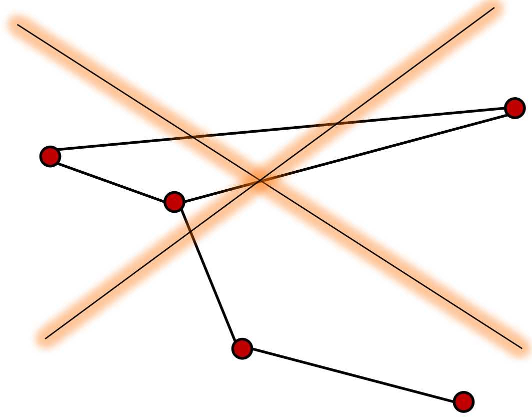
Yury Makarychev
TTIC

Minimum Spanning Tree

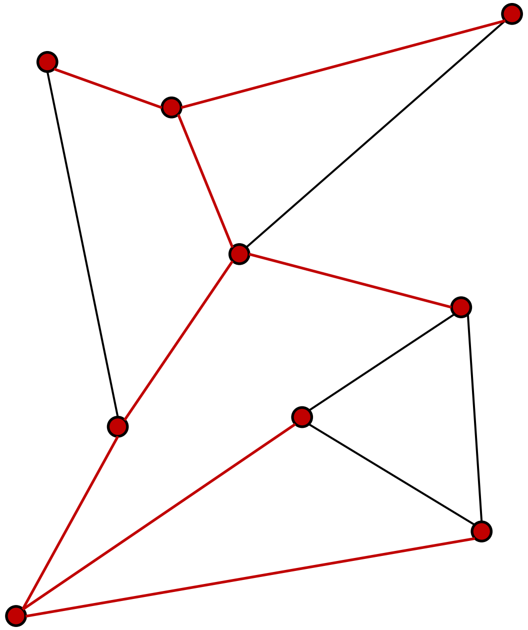
Spanning Tree

- Given a connected graph $G = (V, E, w)$
  - V is the set of vertices
  - E is the set of edges
  - each edge $e \in E$ has weight/length $w_e > 0$
- Recall:
  - A tree is a connected graph without cycles.





Spanning Tree

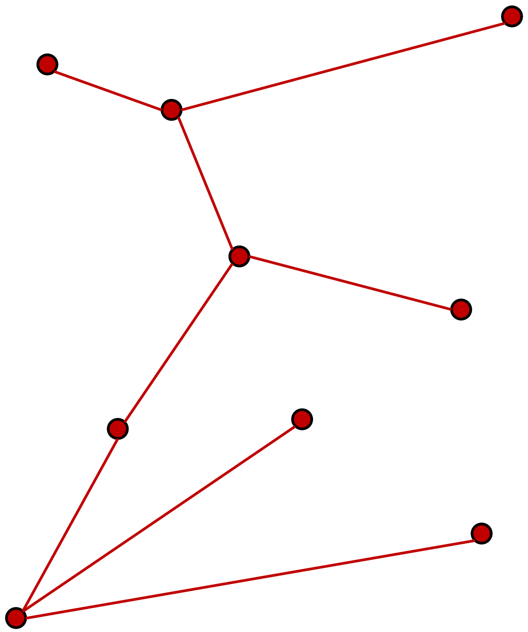


- Given a connected graph $G = (V, E, w)$
  - V is the set of vertices
  - E is the set of edges
  - each edge $e \in E$ has weight/length $w_e > 0$
- Recall:
  - A tree is a connected graph without cycles.
  - T is a spanning tree in G if
    - T is a subgraph of G
    - T is a tree
    - T covers all vertices of G: each vertex of G is also a vertex of T

Spanning Tree

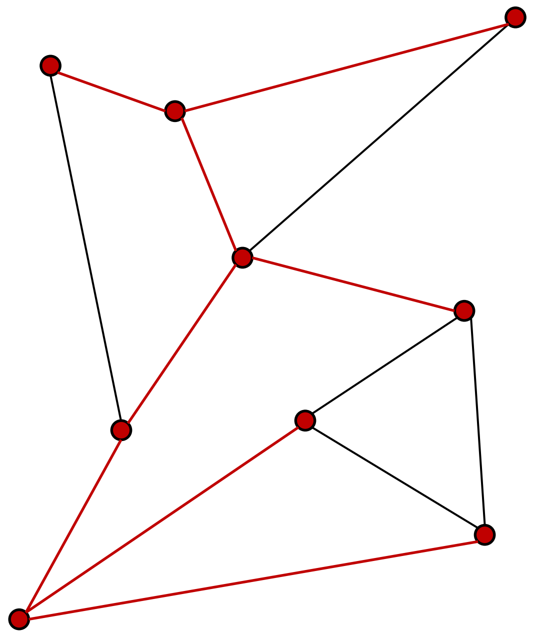
- Recall:
  - A tree is a connected graph without cycles.
  - T is a spanning tree in G if
    - T is a subgraph of G
    - T is a tree
    - each vertex of G is also a vertex of T

- A graph H on k vertices is a tree if and only if
  a. it is connected and
  b. it has exactly $k-1$ edges.

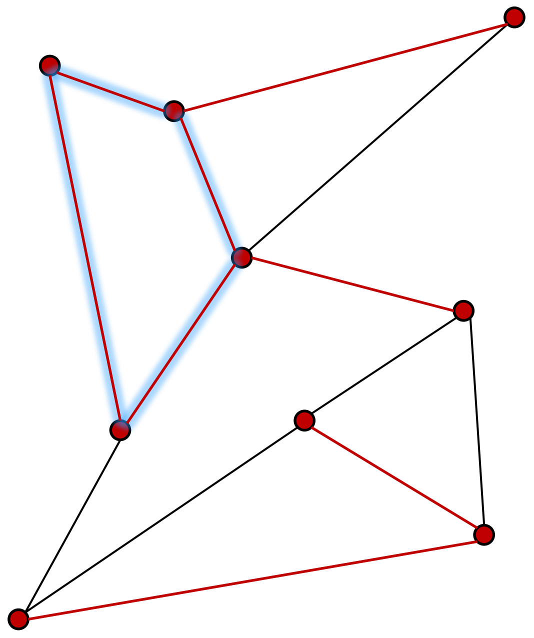


9 vertices and 8 edges in the red tree

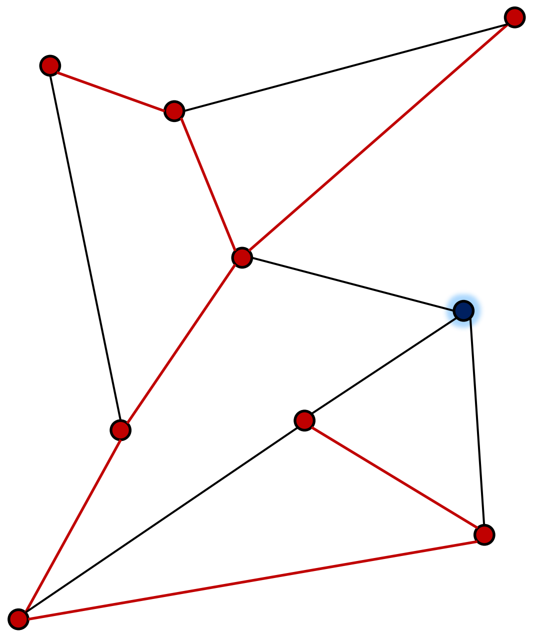
Examples and non-Examples



Spanning tree

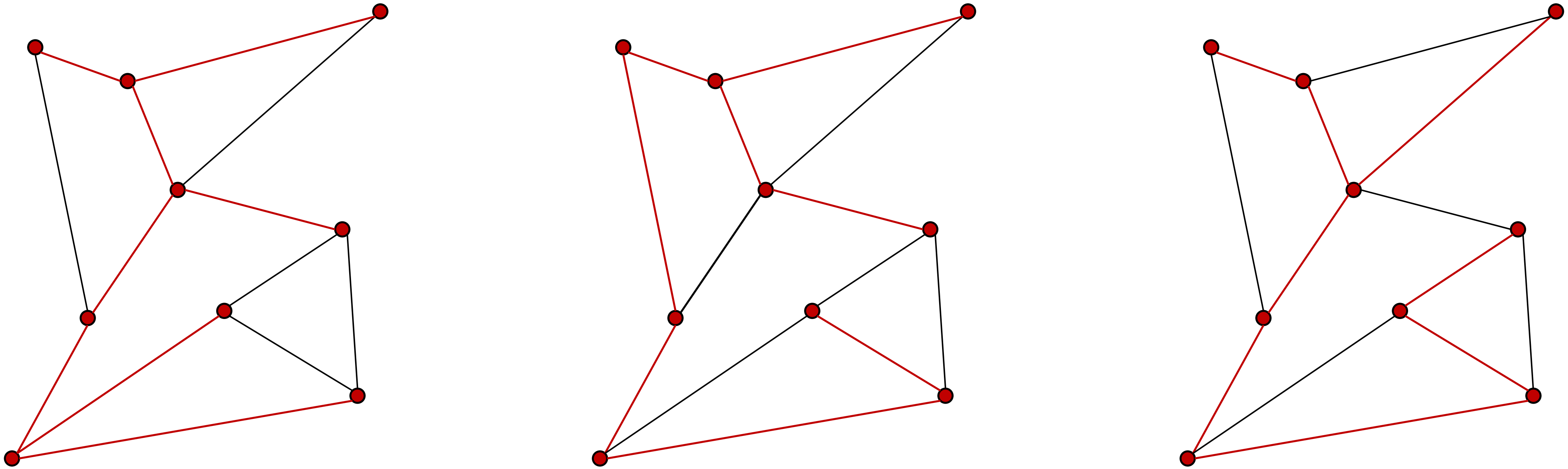


Not a tree: has a cycle and disconnected



Not a spanning subgraph: one vertex is not covered

Many Spanning Trees



Minimum Spanning Tree

- ➢ Given a connected graph $G = (V, E, w)$

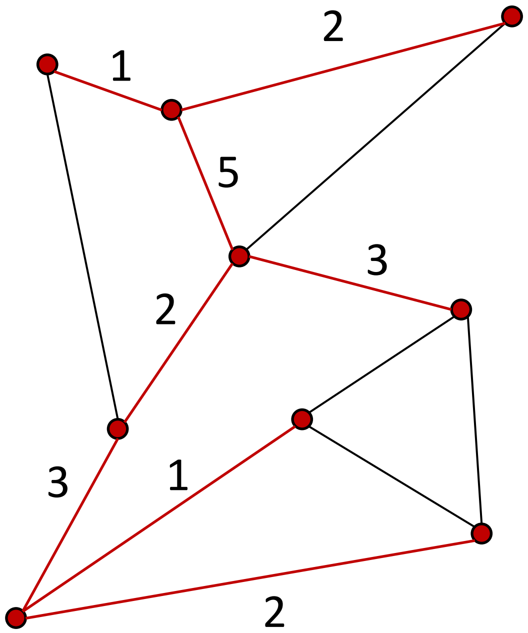
The weight of a spanning tree T is the total weight of its edges.

A minimum spanning tree (MST) is a spanning tree of minimum weight.

- ➢ Goal: find an MST

Two standard algorithms: Prim's and Kruskal's

Both algorithms are greedy. We will study Prim's algorithm.



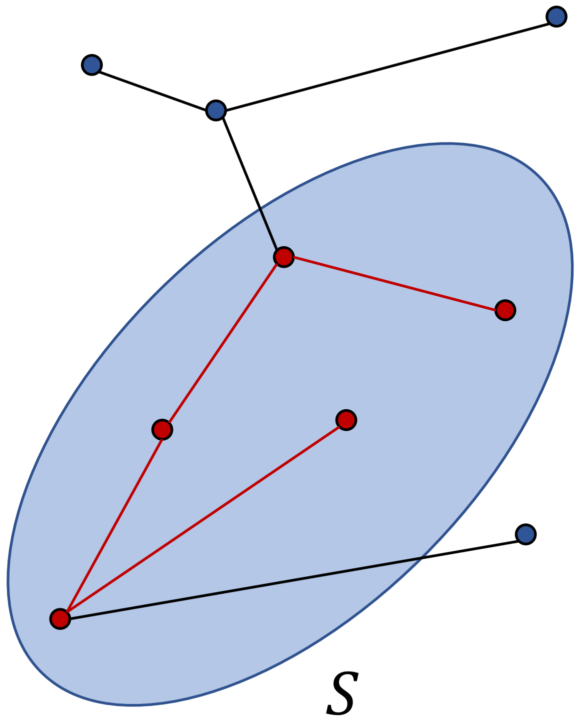
The weight of the red spanning tree is $1 + 2 + 5 + 2 + 3 + 3 + 1 + 2 = 19$

Induced subgraph

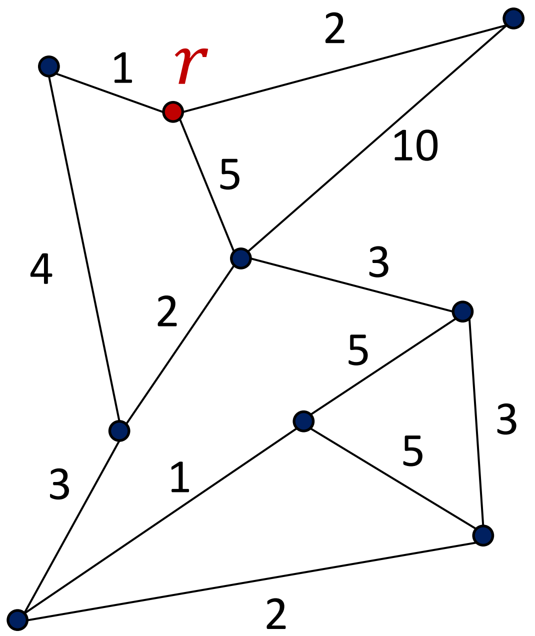
- Let H be a graph and S be a subset of its vertices.

$H[S]$ is a subgraph on S that contains all edges of H that lie within S and no other edges.

$H[S]$ is the subgraph of H induced by S.

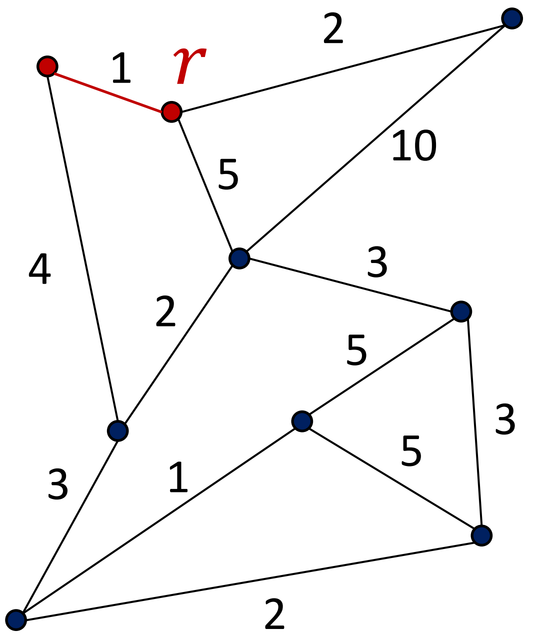


Prim's algorithm



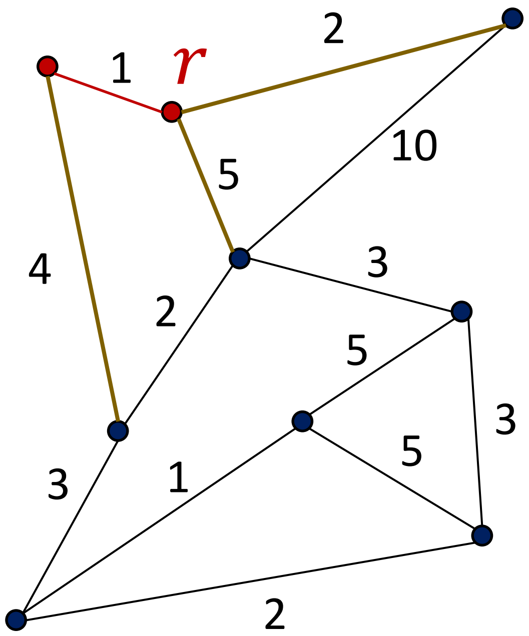
- Choose an arbitrary start vertex r. Let $S = \{r\}$, T be a single-vertex graph on S.

Prim's algorithm



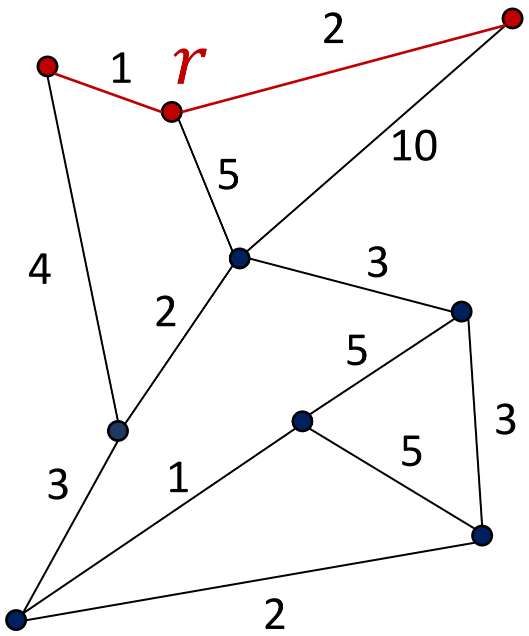
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.

Prim's algorithm



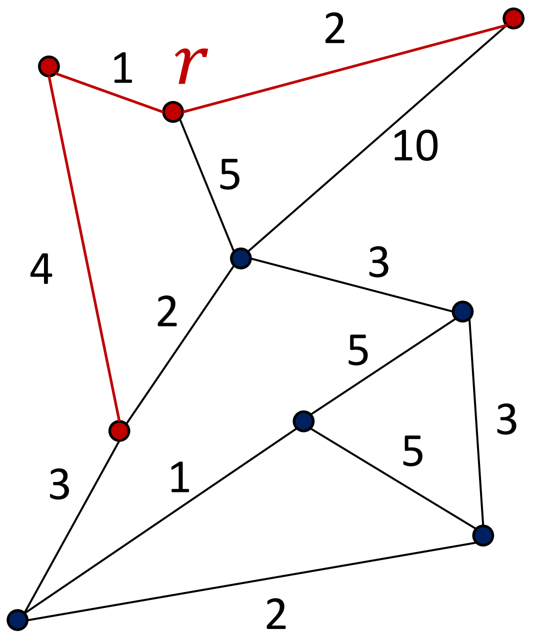
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Q: which edge should we add?

Prim's algorithm



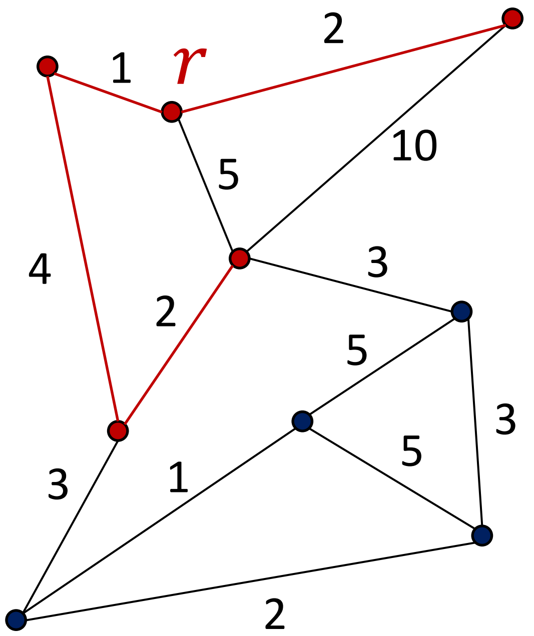
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



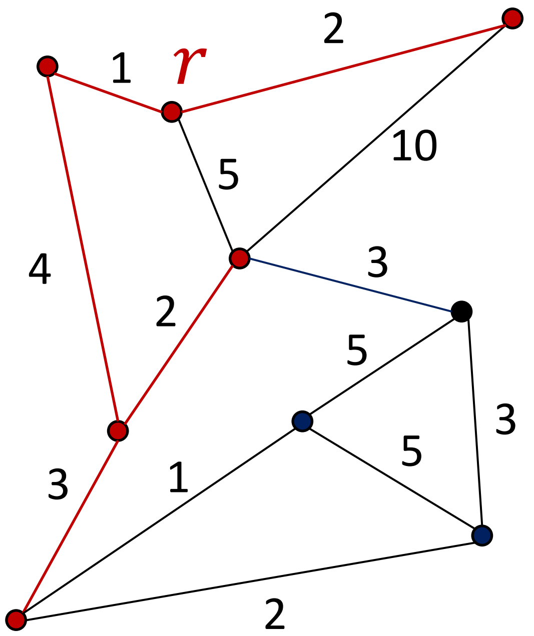
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



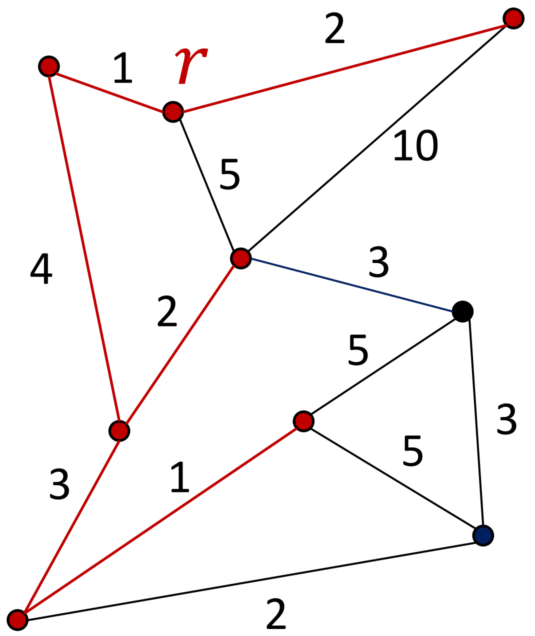
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



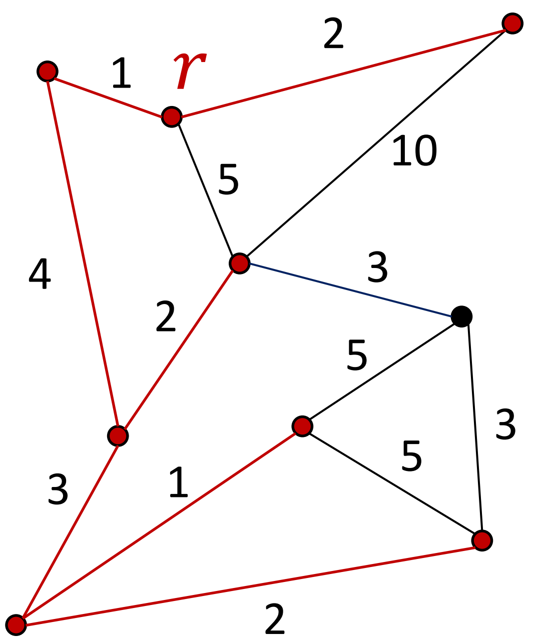
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



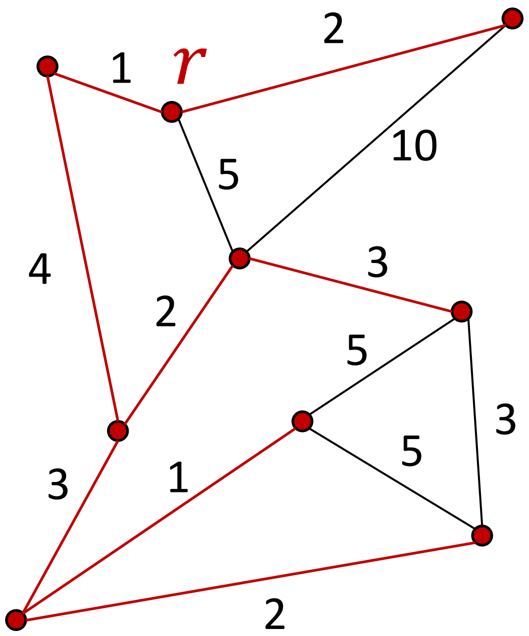
- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

Prim's algorithm



Done!

- Choose an arbitrary start vertex r. Let $S = \{r\}$. T be a single-vertex graph on S.
- Find a shortest edge $e = (r, u)$ incident on r. Add u to S and e to T.
- Consider edges leaving S. Add a shortest among them to T.
- Repeat until all vertices are in S.

TO DO items

- Prove that the algorithm finds a spanning tree.
- Prove that the spanning tree is a minimum spanning tree.
- Discuss how to implement the algorithm and find its running time.

T is a spanning tree in G

Let S_i be the value of S before iteration i starts.
Let T_i be the value of T before iteration i starts.

➢ T_i is a tree on S_i

At each iteration, we add one vertex to S_i and T_i and one edge to T_i.

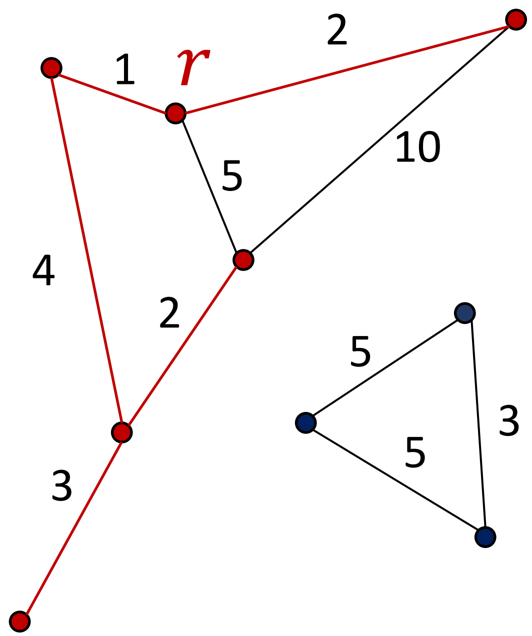
Thus, T_i is a subgraph on i vertices with $i - 1$ edges.

By construction, T_i is connected.

T_i is a tree

Q: Is it possible that there is no edge leaving S before the algorithm terminates?

T is a spanning tree in G



Algorithm:
Consider all edges leaving S.
Add a shortest among them to T.

The algorithm is unable to proceed, since there are no edges leaving S (red vertices).

Optimality

Exchange argument: we prove that our solution is compatible with some optimal solution throughout the execution of the algorithm.

Prove by induction on i:

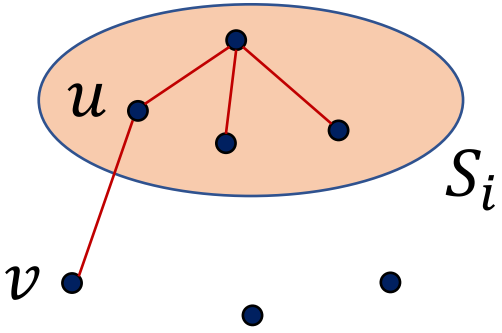
- For every i, there is an MST T' s.t. $T_i = T'[S_i]$
- I.e., T_i can be extended to a minimum spanning tree.

Base case: T_1 is a tree on r. Every minimum spanning tree extends it.

Optimality

Induction step: Assume that $T_i = T'[S_i]$. Prove that $T_{i+1} = T''[S_{i+1}]$ (where T' and T'' are MSTs).

Let (u, v) be the edge that the algorithm adds at iteration i. ($u \in S$, $v \notin S$)

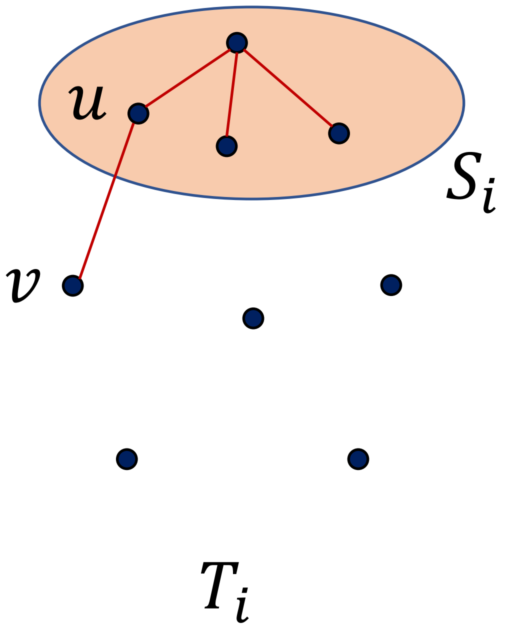


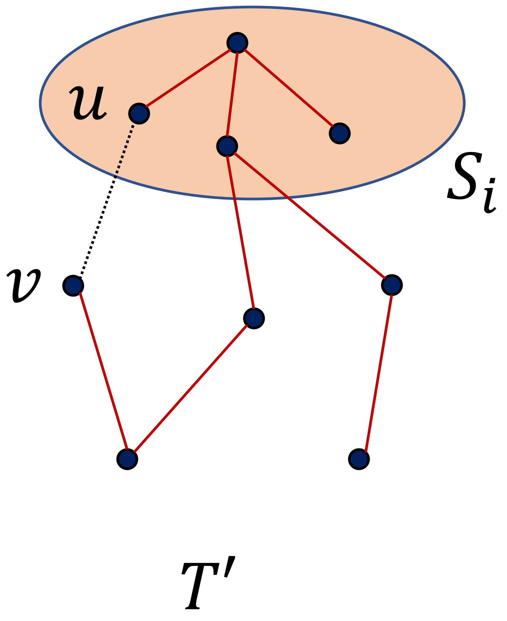
Two cases

- ✓ $(u, v) \in T'$ (easy)
- $(u, v) \notin T'$

Case: $(u, v) \notin \; T'$

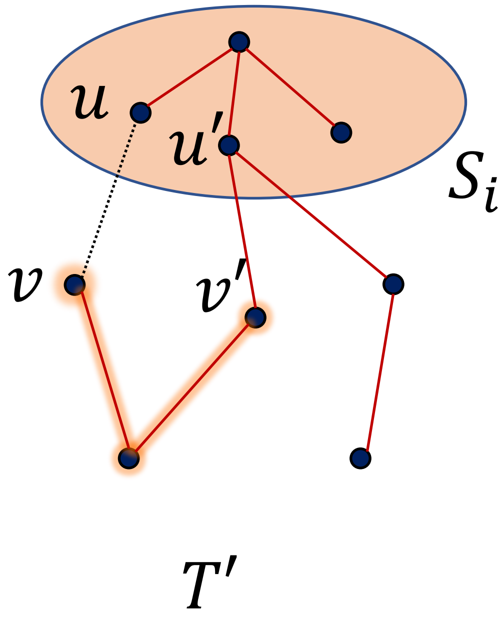
Induction step: Assume that $T_i = T'[S_i]$. Prove that $T_{i+1} = T''[S_{i+1}]$





Case: $(u, v) \notin T'$

Induction step: Assume that $T_i = T'[S_i]$. Prove that $T_{i+1} = T''[S_{i+1}]$

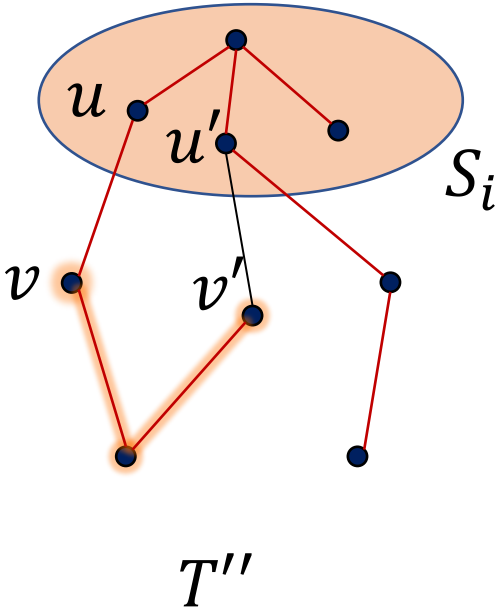


Let H be the connected component of $T' \setminus S_i$ that contains v.

Since T' is connected, H must be connected to S_i with some edge (u', v').

Case: $(u, v) \notin T'$

Induction step: Assume that $T_i = T'[S_i]$. Prove that $T_{i+1} = T''[S_{i+1}]$



Let $T'' = T + (u, v) - (u', v')$.

T'' is consistent with T_i.

T'' is connected
has n vertices and $n-1$ edges $\Big\}$ $\Rightarrow T''$ is a spanning tree

$$w(T'') = w(T') + w(u, v) - w(u', v') \leq w(T')$$

Optimality

Proved:

For every i, there is an MST T' s.t. $T_i = T'[S_i]$

Are we done?

Is T necessarily an MST?

Implementation

Priority Queue

- Q stores elements
- each element x has an associated number p_x, called its key or priority

Methods

`(extract min)` extract the element with least p_x (highest priority)

`(add element)` given y and p_y, add y to Q and set its priority to p_y

`(decrease key)` given y and $p'_y \leq p_y$, change the priority of y to p'_y

Implementation

There are various implementations of priority queues (e.g. using binary heaps). In many of them all queue operations take $O(\log n)$ time.

- Q stores elements
- each element x has an associated number p_x, called its key or priority

Methods

`(extract min)` extract the element with least p_x (highest priority)

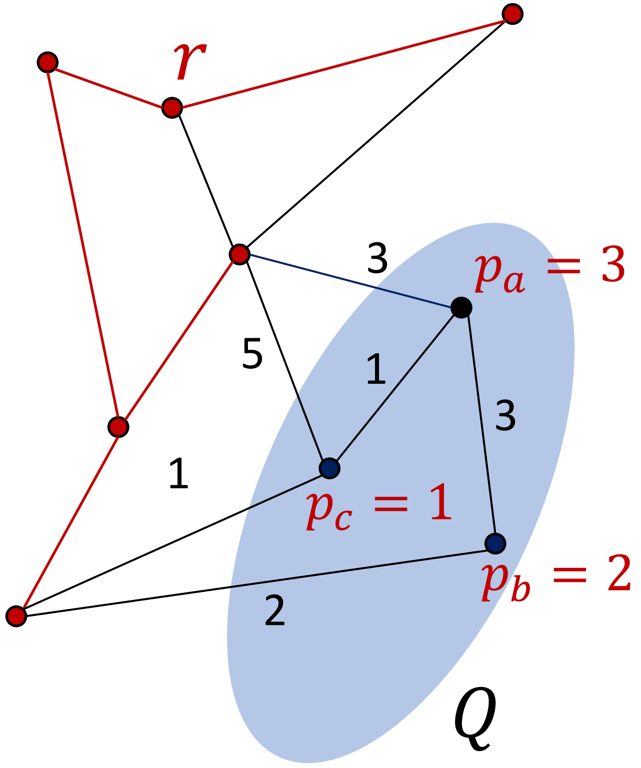
`(add element)` given y and p_y, add y to Q and set its priority to p_y

`(decrease key)` given y and $p'_y \leq p_y$, change the priority of y to p'_y

Implementation

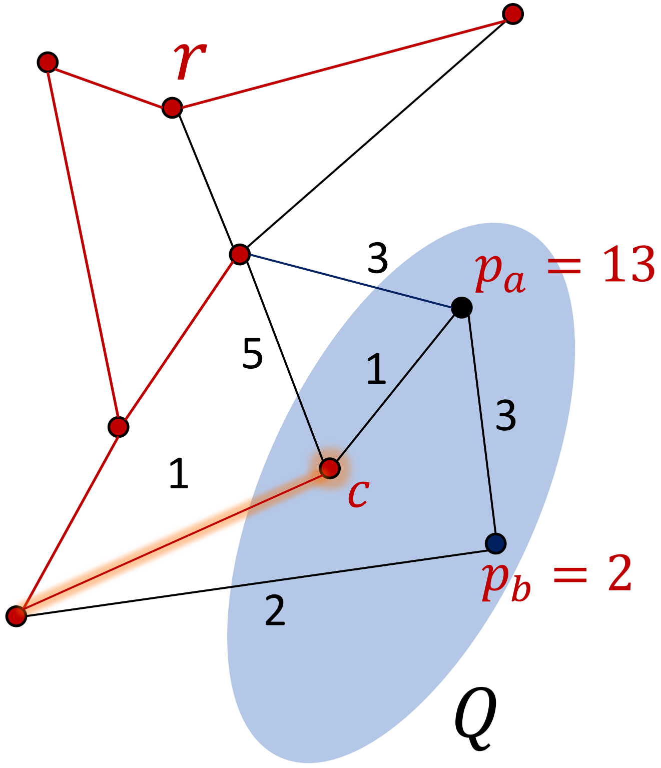
- Choose an arbitrary vertex r in T
- Initialize a priority queue Q
- Q stores elements that are not in S; that is, vertices that are to be processed
- Add all vertices u other than r to Q.
- For u in Q, p_u will the length of the shortest edge that connects u to a vertex in S
- Initially, set $p_u = \begin{cases} w_{ru}, & \text{if } (r,u) \in E \\ +\infty, & \text{otherwise} \end{cases}$

c has the highest priority in the queue



Implementation

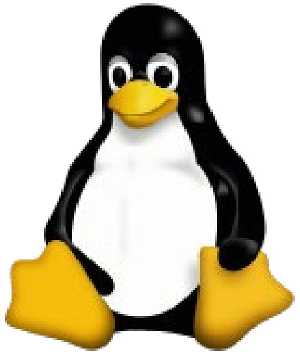
- while Q is not empty
  - extract a vertex u from Q with the least value of p_u
  - remove u from Q, add u to T
  - add the shortest edge that connects u with S to T
  - for all vertices v adjacent to u
    - if v is in Q, $p_v = \min(p_v, w_{uv})$



- Implementation details: in addition to the value of p_u, store the shortest edge from u to S
- $O(m)$ queue and other operations ⇒ the running time is $O(m \log n)$

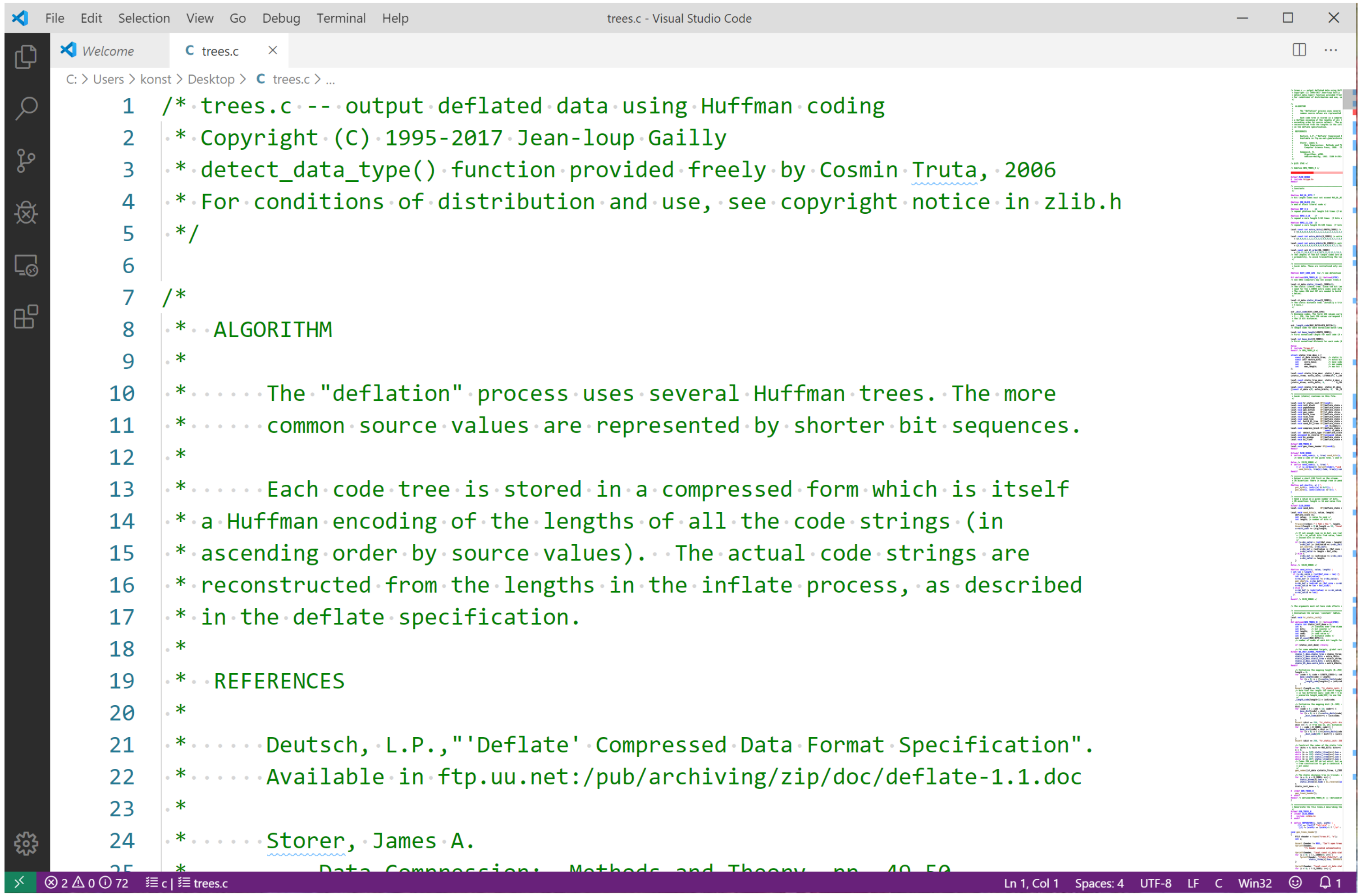
Huffman Coding

What do these products have in common?



APACHE
HTTP SERVER PROJECT

They use zlib and Huffman Coding



```
/* trees.c -- output deflated data using Huffman coding
 * Copyright (C) 1995-2017 Jean-loup Gailly
 * detect_data_type() function provided freely by Cosmin Truta, 2006
 * For conditions of distribution and use, see copyright notice in zlib.h
 */

/*
 *  ALGORITHM
 *
 *      The "deflation" process uses several Huffman trees. The more
 *      common source values are represented by shorter bit sequences.
 *
 *      Each code tree is stored in a compressed form which is itself
 * a Huffman encoding of the lengths of all the code strings (in
 * ascending order by source values).  The actual code strings are
 * reconstructed from the lengths in the inflate process, as described
 * in the deflate specification.
 *
 *  REFERENCES
 *
 *      Deutsch, L.P.,"'Deflate' Compressed Data Format Specification".
 *      Available in ftp.uu.net:/pub/archiving/zip/doc/deflate-1.1.doc
 *
 *      Storer, James A.
 *          Data Compression:  Methods and Theory, pp. 49-50.
```

Character encoding

There are many languages and scripts (below I listed only a very small fraction of them)

العربية
বাংলা
Български
Bosanski
Català
Čeština
Dansk
Deutsch
Eesti
Ελληνικά

Español
Esperanto
Euskara
فارسی
Français
Galego
한국어
Hrvatski
Italiano
עברית

ქართული
Latviešu
Lietuvių
Magyar
Македонски
Bahasa Melayu
Nederlands
日本語
Norsk bokmål
Norsk nynorsk

Polski
Português
Română
Русский
Simple English
Slovenčina
Slovenščina
Српски / srpski
Srpskohrvatski / српскохрватски

Suomi
Svenska
ไทย
Türkçe
Українська
Tiếng Việt
中文

We want to design an efficient way to encode characters with 0s and 1s

How do we encode characters?

- All data in computers is stored as 0’s and 1’s.
- Characters are encoded with 0’s and 1’s.
- Standard formats:
  - Extended ASCII and Unicode (UTF-8, UTF-16, and UTF-32).
  - Extended ASCII and UTF-32 are fixed-length encoding.
  - UTF-8 and UTF-16 are variable-length encodings.

➢ How do we design an efficient encoding scheme?

Character encoding

- We are given an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$.
- Want to encode texts written in Σ with binary strings.

- Choose an encoding function $f : \Sigma \to \{0,1\}^*$ (where $\{0,1\}^*$ denotes the set of binary strings)
- each character σ_i is encoded with codeword $f(\sigma_i)$
- to encode a string $\sigma_{a_1}\sigma_{a_2} \dots \sigma_{a_k}$
  1. we encode individual characters
  2. then concatenate the obtained codewords

$$f(\sigma_{a_1})f(\sigma_{a_2}) \dots f(\sigma_{a_k})$$

Character encoding

- ➢ We are given an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$.
- ➢ Want to encode texts written in Σ with binary strings.

- Choose an encoding function $f: \Sigma \to \{0,1\}^*$ (where $\{0,1\}^*$ denotes the set of binary strings)
- each character σ_i is encoded with codeword $f(\sigma_i)$
- to encode a string $\sigma_{a_1}\sigma_{a_2} \dots \sigma_{a_k}$
  1. we encode individual characters
  2. then concatenate the obtained codewords

$$f(\sigma_{a_1})f(\sigma_{a_2}) \dots f(\sigma_{a_k})$$

Example

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

$abc \mapsto 01011$

$baba \mapsto 100100$

$abac \mapsto 010011$

codewords: 0, 10,11

Character encoding

Can we choose arbitrary codewords?

No! We need to ensure that every encoded message has a unique decoding?

Example

$\Sigma = \{a, b, c\}$

- $f: a \mapsto 0$
- $f: b \mapsto 1$
- $f: c \mapsto 10$

Does message 0110 encode $abba$ or abc?

Uniquely Decodable Encoding

We say that encoding f is uniquely decodable if every binary string has at most 1 decoding. That is,

$$f(s_1) \neq f(s_2)$$

for every two different strings s_1 and s_2 over alphabet Σ.

- The simplest example of a uniquely decodable encoding is a fixed-length code. All codewords have the same length. E.g., in extended ASCII all codewords have length 8:

$$010010010010101001010101010101010101010000101000$$

Uniquely Decodable Encoding

- The simplest example of a uniquely decodable encoding is a fixed-length code. All codewords have the same length. E.g., in extended ASCII all codewords have length 8:

$$01001001001010100101010101010101010101000101000$$

- Another example of uniquely decodable code is a prefix code. In a prefix code no codeword is a prefix of another.

prefix

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: 0, 10,11

not prefix

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 1$

$f: c \mapsto 10$

codewords: 0, 1,10

1 is a prefix of 10

Decoding prefix codes

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: $0, 10{,}11$

$$0100111100101010010101010101010101010101000101$$

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: $0, 10, 11$

Decoding prefix codes

$01001111001010100101010101010101010101000101$
a

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: $0, 10{,}11$

$0100111100101010010101010101010101010101000101$

a

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: 0, 10,11

$$01001111001010100101010101010101010101000101$$

$$ab$$

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: 0, 10,11

$010011110010101001010101010101010101010101000101$
$abacc$ …

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

It's easy to prove by induction that the algorithm outputs the only possible decoding of the binary string.

Prefix codes

- Q: are fixed-length codes prefix codes?
- A: yes

Prefix codes

- Q: are there uniquely decodable codes that are not prefix codes?
- A: yes, suffix codes.

$\Sigma = \{a, b, c\}$

$f: a \mapsto 0$

$f: b \mapsto 01$

$f: c \mapsto 11$

codewords: $0, 01, 11$

Optimal Codes

Optimal encoding problem

- Assume that we are given probabilities/frequencies $p_1, \dots, p_n$ with which characters $\sigma_1, \dots, \sigma_n$ appear in texts.
- How many bits do we need to encode a text with the given character frequencies?

$$\mathrm{cost}(f) = p_1|f(\sigma_1)| + \cdots + p_n|f(\sigma_n)|$$

per character of text (here, $|f(\sigma_i)|$ is the length of codeword $f(\sigma_i)$).

The encoding problem: find uniquely decodable f that minimizes $\mathrm{cost}(f)$.

Claim: there is an optimal uniquely decodable code that is a prefix code.

Tree representation

Let f be a prefix code.

- Consider all codewords $C = \{f(\sigma_1), \ldots, f(\sigma_n)\}$.
- Further consider all prefixes P of these codewords (other than codewords themselves).
- Create a binary tree on $P \cup C$
  - The tree is rooted at Λ (that is, empty string)
  - String u has children $u0$ and $u1$, if they are present in $P \cup C$
  - Codewords are leaves of the tree
  - Each leaf $f(\sigma_i)$ is labelled with σ_i

$\Sigma = \{a, b, c\}$

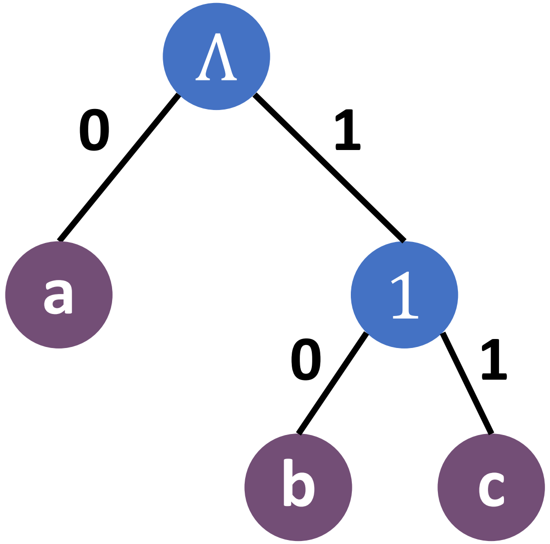
$f: a \mapsto 0$

$f: b \mapsto 10$

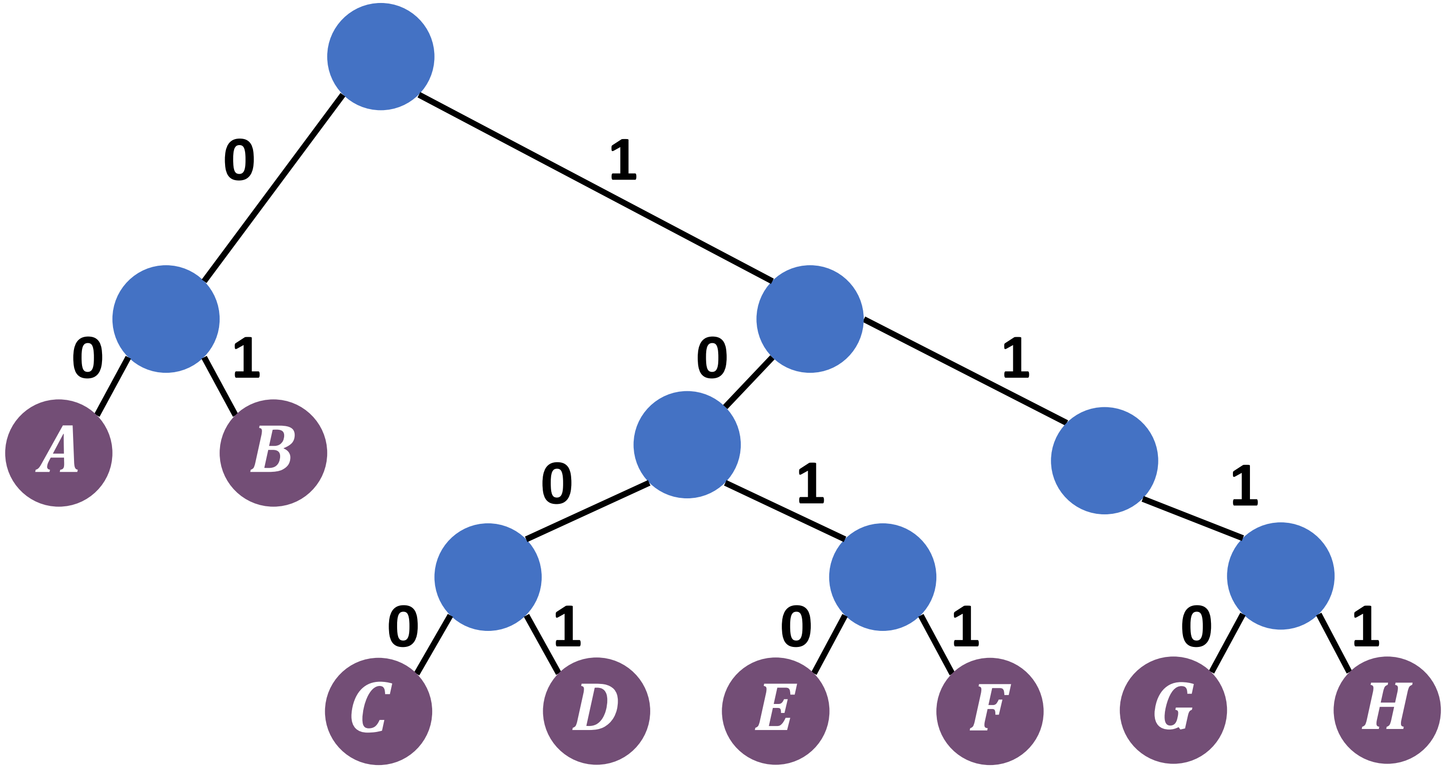
$f: c \mapsto 11$

codewords: $0, 10, 11$

prefixes: $\Lambda, 1$

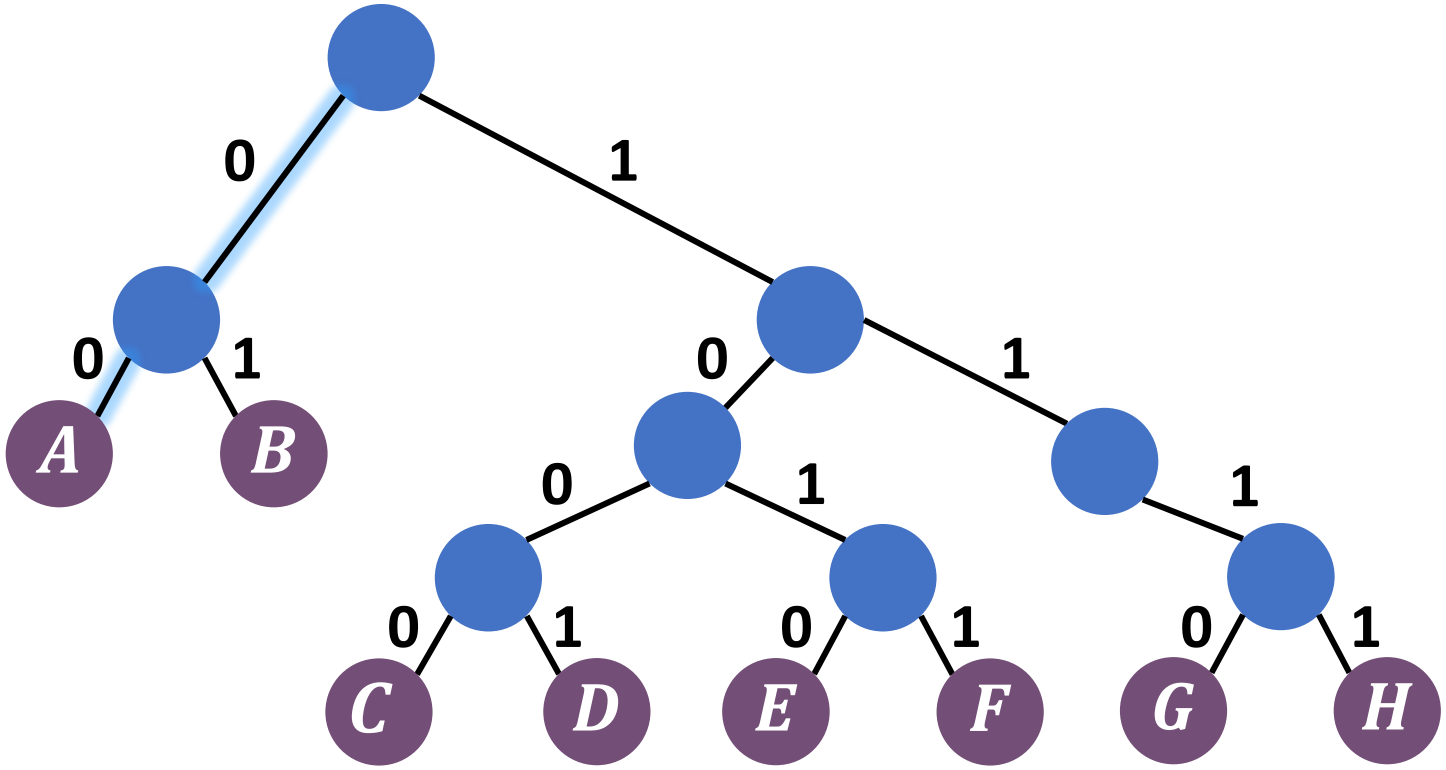


Tree Representation



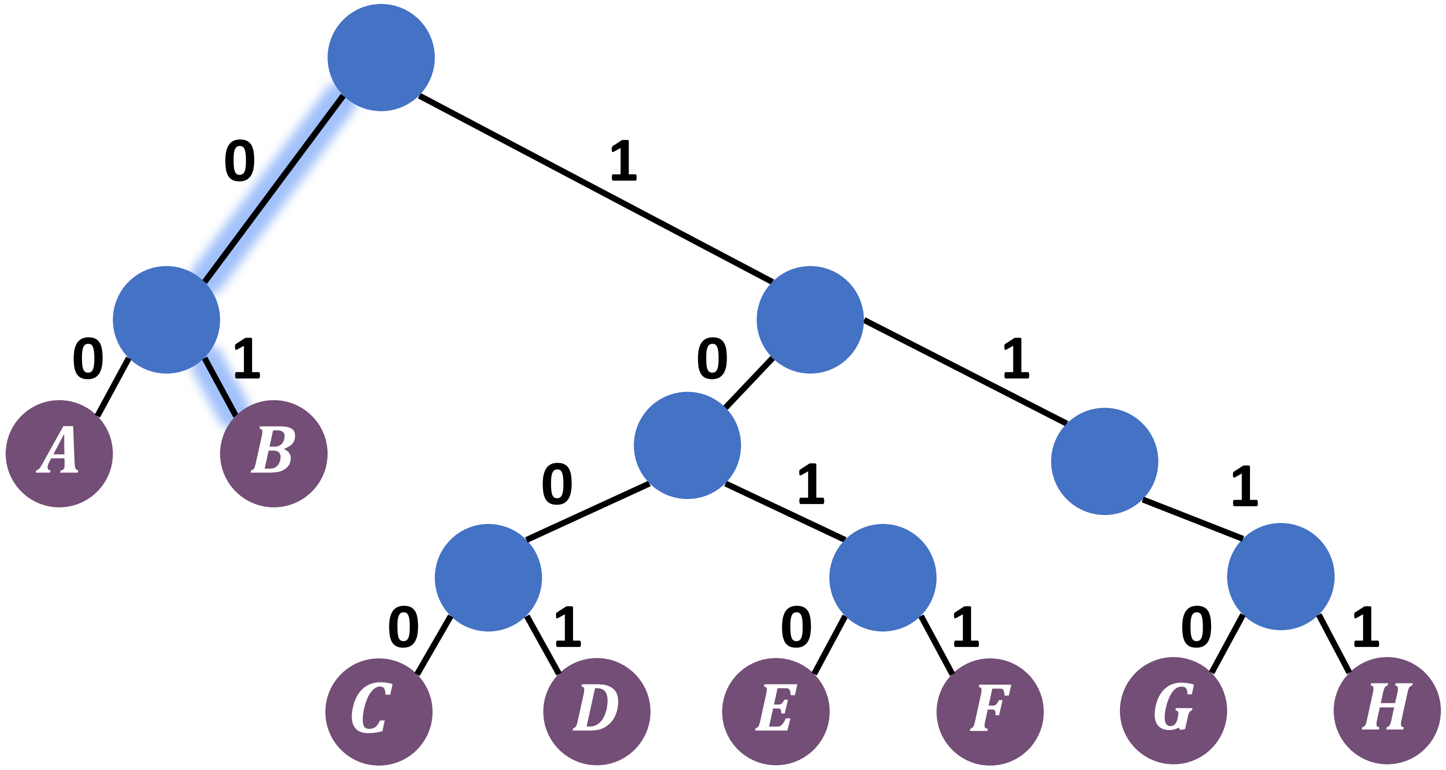
Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



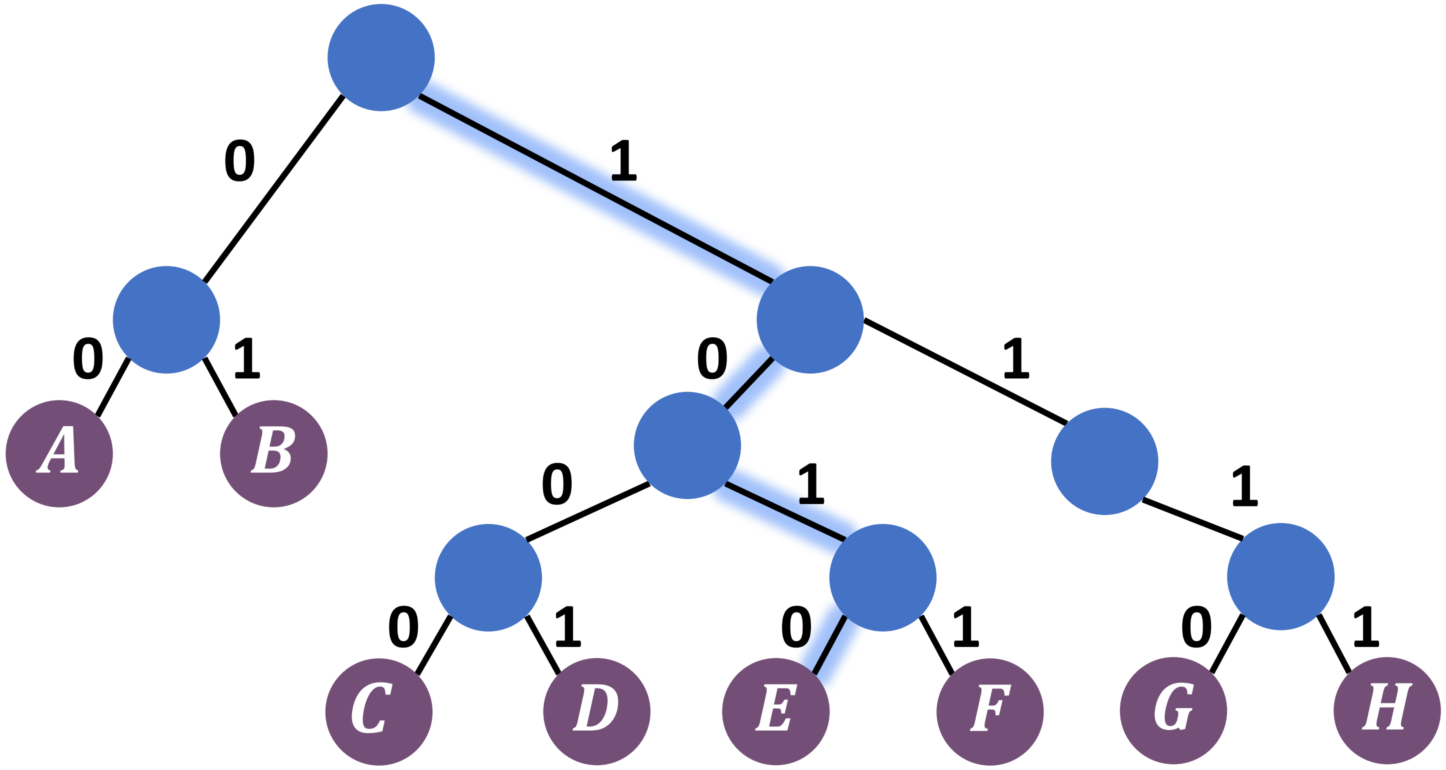
Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation

- ➢ Each prefix code defines a prefix tree.

Consider two vertices (binary strings) in a prefix tree: B_1 and B_2.

B_1 is a prefix of B_2 if and only if B_1 is an ancestor of B_2

- ➢ Given a prefix tree, consider the leaves and their labels. They define a code.

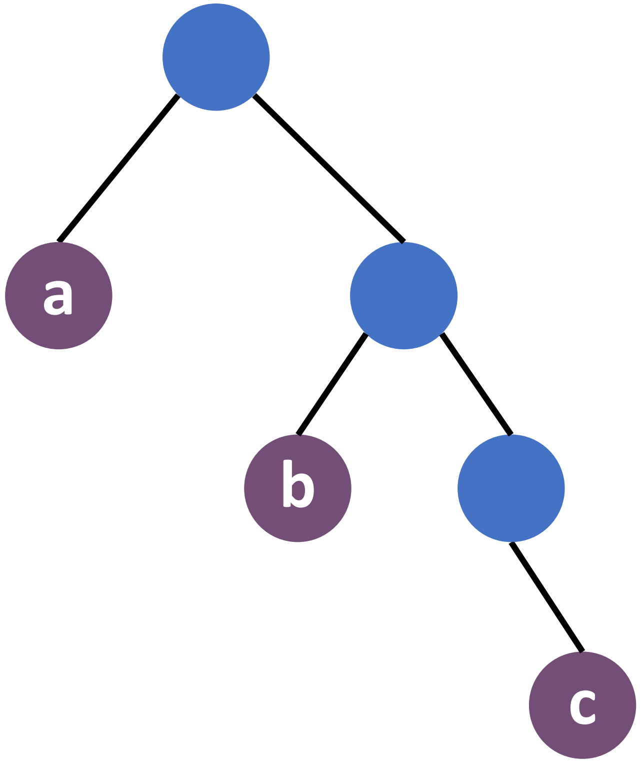
Since no leaf is an ancestor of another, each prefix tree defines a prefix code.

Cost

$$\mathrm{cost}(f) = \sum_i p_i |f(\sigma_i)| = \sum_i p_i \,\mathrm{depth}(f(\sigma_i))$$

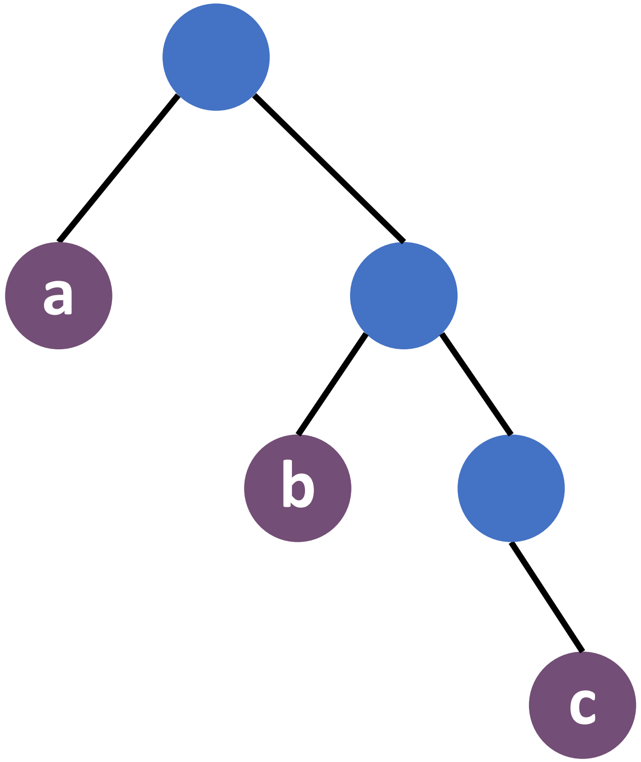
Optimal Binary Tree

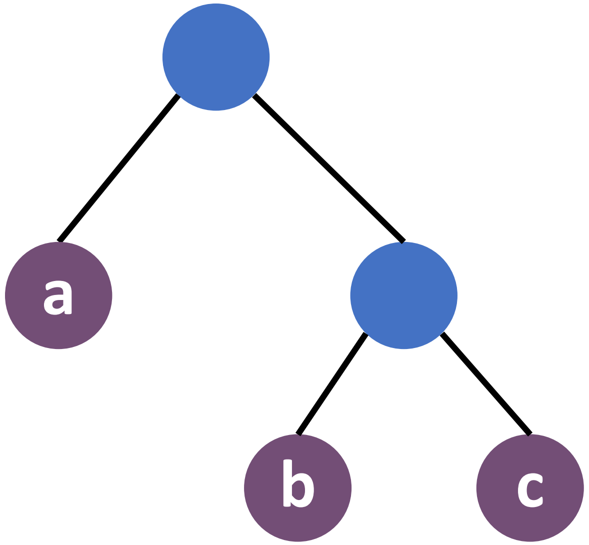
Can this tree be an optimal tree for some code and some set of p_i?



Optimal Binary Tree

Can this tree be an optimal tree for some code and some set of p_i? No!





Optimal Binary Tree

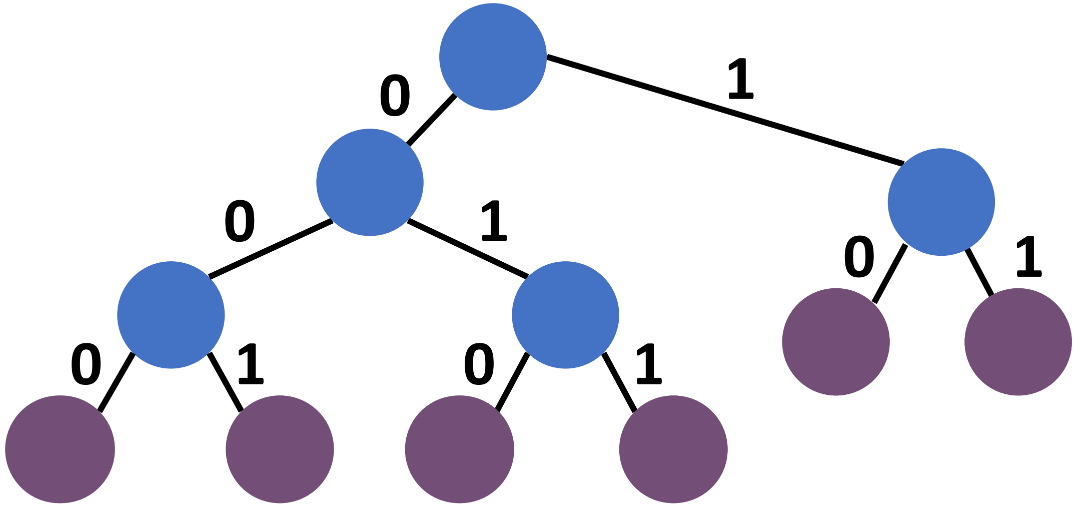
Claim:

- An optimal tree is a full binary tree:
  - all internal nodes (not leaves) have exactly two children.
  - in other words, each node has 0 or 2 children.

Optimal Labeling?

- **Q:** Suppose we are given frequencies $p_1, \dots, p_n$ and an optimal prefix tree. But the labels of the leaves (codewords) are hidden.

How can we find an optimal labeling of leaves with characters σ_i?



Optimal Labeling?

- Q: Suppose we are given frequencies $p_1, \dots, p_n$ and an optimal prefix tree. But the labels of the leaves (codewords) are hidden.

How can we find an optimal labeling of leaves with characters σ_i?

A: Sort all codewords by their length / depth in the tree.
Assign shorter codewords to more frequent letters.

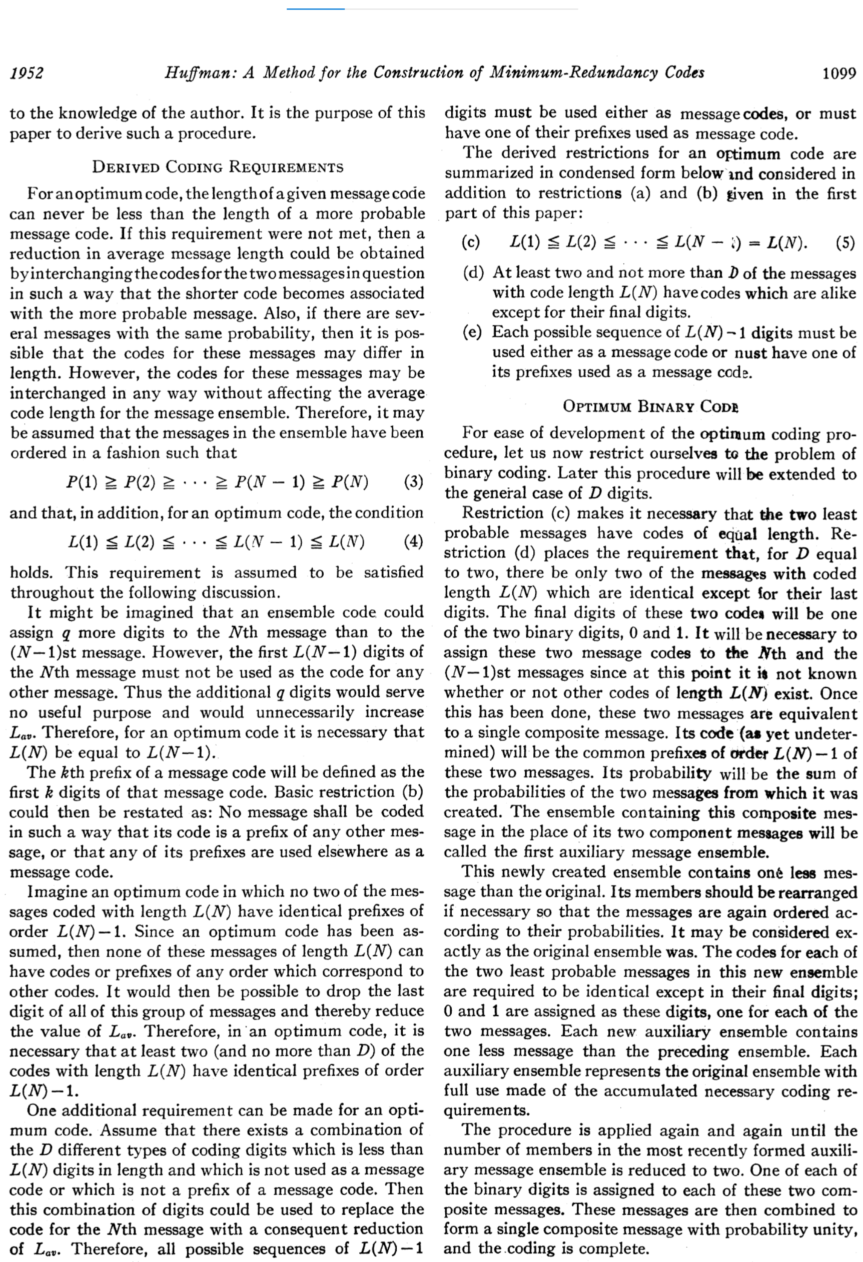
Proof: Assume that leaves at depths d_1 and d_2 are labeled with characters with frequencies $p_1 > p_2$, and $d_1 > d_2$.

Swap the labels of the leaves. We get a better encoding:

$$p_1 d_1 + p_2 d_2 > p_1 d_2 + p_2 d_1 \text{ since } (p_1 - p_2)(d_1 - d_2) > 0.$$

Huffman coding

- Huffman proposed a greedy algorithms for constructing prefix codes in 1952.



to the knowledge of the author. It is the purpose of this paper to derive such a procedure.

Derived Coding Requirements

For an optimum code, the length of a given message code can never be less than the length of a more probable message code. If this requirement were not met, then a reduction in average message length could be obtained by interchanging the codes for the two messages in question in such a way that the shorter code becomes associated with the more probable message. Also, if there are several messages with the same probability, then it is possible that the codes for these messages may differ in length. However, the codes for these messages may be interchanged in any way without affecting the average code length for the message ensemble. Therefore, it may be assumed that the messages in the ensemble have been ordered in a fashion such that

$$P(1) \geqq P(2) \geqq \cdots \geqq P(N-1) \geqq P(N) \qquad (3)$$

and that, in addition, for an optimum code, the condition

$$L(1) \leqq L(2) \leqq \cdots \leqq L(N-1) \leqq L(N) \qquad (4)$$

holds. This requirement is assumed to be satisfied throughout the following discussion.

It might be imagined that an ensemble code could assign q more digits to the Nth message than to the $(N-1)$st message. However, the first $L(N-1)$ digits of the Nth message must not be used as the code for any other message. Thus the additional q digits would serve no useful purpose and would unnecessarily increase L_{av}. Therefore, for an optimum code it is necessary that $L(N)$ be equal to $L(N-1)$.

The kth prefix of a message code will be defined as the first k digits of that message code. Basic restriction (b) could then be restated as: No message shall be coded in such a way that its code is a prefix of any other message, or that any of its prefixes are used elsewhere as a message code.

Imagine an optimum code in which no two of the messages coded with length $L(N)$ have identical prefixes of order $L(N)-1$. Since an optimum code has been assumed, then none of these messages of length $L(N)$ can have codes or prefixes of any order which correspond to other codes. It would then be possible to drop the last digit of all of this group of messages and thereby reduce the value of L_{av}. Therefore, in an optimum code, it is necessary that at least two (and no more than D) of the codes with length $L(N)$ have identical prefixes of order $L(N)-1$.

One additional requirement can be made for an optimum code. Assume that there exists a combination of the D different types of coding digits which is less than $L(N)$ digits in length and which is not used as a message code or which is not a prefix of a message code. Then this combination of digits could be used to replace the code for the Nth message with a consequent reduction of L_{av}. Therefore, all possible sequences of $L(N)-1$ digits must be used either as message codes, or must have one of their prefixes used as message code.

The derived restrictions for an optimum code are summarized in condensed form below and considered in addition to restrictions (a) and (b) given in the first part of this paper:

$$\text{(c)} \qquad L(1) \leqq L(2) \leqq \cdots \leqq L(N-1) = L(N). \qquad (5)$$

(d) At least two and not more than D of the messages with code length $L(N)$ have codes which are alike except for their final digits.

(e) Each possible sequence of $L(N)-1$ digits must be used either as a message code or must have one of its prefixes used as a message code.

Optimum Binary Code

For ease of development of the optimum coding procedure, let us now restrict ourselves to the problem of binary coding. Later this procedure will be extended to the general case of D digits.

Restriction (c) makes it necessary that the two least probable messages have codes of equal length. Restriction (d) places the requirement that, for D equal to two, there be only two of the messages with coded length $L(N)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the Nth and the $(N-1)$st messages since at this point it is not known whether or not other codes of length $L(N)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $L(N)-1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages will be called the first auxiliary message ensemble.

This newly created ensemble contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. The codes for each of the two least probable messages in this new ensemble are required to be identical except in their final digits; 0 and 1 are assigned as these digits, one for each of the two messages. Each new auxiliary ensemble contains one less message than the preceding ensemble. Each auxiliary ensemble represents the original ensemble with full use made of the accumulated necessary coding requirements.

The procedure is applied again and again until the number of members in the most recently formed auxiliary message ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite message with probability unity, and the coding is complete.

To be continued