Design and Analysis of Algorithms

Application of Greedy Algorithms

1. Huffman Coding
2. Single Source Shortest Path Problem
   - Dijkstra's Algorithm
3. Minimal Spanning Tree
   - Kruskal's Algorithm
   - Prim's Algorithm

Applications of Greedy Algorithms

Huffman coding

- Proof of Huffman algorithm

Single source shortest path algorithm

- Dijkstra algorithm and correctness proof

Minimal spanning trees

- Prim's algorithm
- Kruskal's algorithm

Outline

1. Huffman Coding
2. Single Source Shortest Path Problem
   - Dijkstra's Algorithm
3. Minimal Spanning Tree
   - Kruskal's Algorithm
   - Prim's Algorithm

Motivation of Coding

Let Γ be an alphabet of n symbols, the frequency of symbol i is f_i. Clearly, $\sum_{i=1}^{n} f_i = 1$.

In practice, to facilitate transmission in the digital world (improve efficiency and robustness), we use coding to encode symbols (characters in a file) into codewords, then transmit.

$$\boxed{\Gamma} \xrightarrow{\text{encoding}} \boxed{C} \xrightarrow{\text{transmit}} \boxed{C} \xrightarrow{\text{decoding}} \boxed{\Gamma}$$

What is a good coding scheme?

- efficiency: efficient encoding/decoding & low bandwidth
- other nice properties about robustness, e.g., error-correcting

Fix-Length vs. Variable-Length Coding

Two approaches of encoding a source (Γ, F)

- Fix-length: the length of each codeword is a fixed number
- Variable-length: the length of each codeword could be different

Fix-Length vs. Variable-Length Coding

Two approaches of encoding a source (Γ, F)

- Fix-length: the length of each codeword is a fixed number
- Variable-length: the length of each codeword could be different

Fix-length coding seems neat. Why bother to introduce variable-length coding?

Fix-Length vs. Variable-Length Coding

Two approaches of encoding a source (Γ, F)

- Fix-length: the length of each codeword is a fixed number
- Variable-length: the length of each codeword could be different

Fix-length coding seems neat. Why bother to introduce variable-length coding?

- If each symbol appears with same frequency, then fix-length coding is good.
- If (Γ, F) is non-uniform, we can use less bits to represent more frequent symbols $\Rightarrow$ more economical coding $\Rightarrow$ low bandwidth

Average Codeword Length

Let f_i be the frequency of i-th symbol, ℓ_i be its codeword length. Average code length capture the average length of codeword for source (Γ, F)

$$L = \sum_{i=1}^{n} f_i \ell_i$$

Problem. Given a source (Γ, F), finding its optimal encoding (minimize average codeword length).

Average Codeword Length

Let f_i be the frequency of i-th symbol, ℓ_i be its codeword length. Average code length capture the average length of codeword for source (Γ, F)

$$L = \sum_{i=1}^{n} f_i \ell_i$$

Problem. Given a source (Γ, F), finding its optimal encoding (minimize average codeword length).

Wait... There is still a subtle problem here.

Prefix-free Coding

Prefix-free. No codeword cannot be a prefix of another codeword.

prefix code (前缀码) $\rightsquigarrow$ prefix-free code

Prefix-free Coding

Prefix-free. No codeword cannot be a prefix of another codeword.

prefix code (前缀码) $\rightsquigarrow$ prefix-free code

All fixed-length encodings naturally satisfy prefix-free property.

- Prefix-free property is only meaningful for variable-length encoding.

Prefix-free Coding

Prefix-free. No codeword cannot be a prefix of another codeword.

prefix code (前缀码) $\rightsquigarrow$ prefix-free code

All fixed-length encodings naturally satisfy prefix-free property.

- Prefix-free property is only meaningful for variable-length encoding.

Application of prefix-free encoding (consider the case of binary encoding)

- Not prefix-free: the coding may not be uniquely decipherable. Out-of-band channel is needed to transmit delimiter.
- Prefix-free: unique decipherable, decoding does not require out-of-band transmit

Ambiguity of Non-Prefix-Free Encoding

Example. Non prefix-free encoding

- $\langle a, 001\rangle$, $\langle b, 00\rangle$, $\langle c, 010\rangle$, $\langle d, 01\rangle$

Decoding of string like 0100001 is ambiguous

- decoding 1: $01 \mid 00 \mid 001 \Rightarrow (d, b, a)$
- decoding 2: $010 \mid 00 \mid 01 \Rightarrow (c, b, d)$

Ambiguity of Non-Prefix-Free Encoding

Example. Non prefix-free encoding

- $\langle a, 001\rangle$, $\langle b, 00\rangle$, $\langle c, 010\rangle$, $\langle d, 01\rangle$

Decoding of string like 0100001 is ambiguous

- decoding 1: $01 \mid 00 \mid 001 \Rightarrow (d, b, a)$
- decoding 2: $010 \mid 00 \mid 01 \Rightarrow (c, b, d)$

Refinement of Problem

How to find an optimal prefix-free encoding?

Tree Representation of Prefix-free Encoding

Any prefix-free encoding can be represented by a binary tree.

- all symbols are at the leaves (this property implies prefix-freeness, since no leaf node could be a prefix of other leaf nodes)
- each codeword is generated by a path from root to leaf, interpreting "left" as 0 and "right" as 1
- codeword length for symbol $i \leftarrow$ depth of leaf node i in the tree

Tree Representation of Prefix-free Encoding

Any prefix-free encoding can be represented by a binary tree.

- all symbols are at the leaves (this property implies prefix-freeness, since no leaf node could be a prefix of other leaf nodes)
- each codeword is generated by a path from root to leaf, interpreting "left" as 0 and "right" as 1
- codeword length for symbol $i \leftarrow$ depth of leaf node i in the tree

Bonus: Decoding is unique

- A string of bits is decoded by starting at the root, reading the string from left to right to move downward along the tree.
- Whenever a leaf is reached, outputting the corresponding symbol and returning to the root.

Simple Fact

An optimal prefix-free encoding corresponds to a *full binary tree*

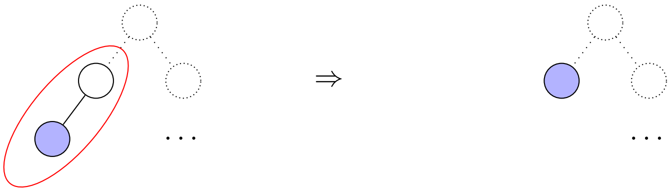
- every node has either zero or two children
- except leaves, each internal node have two children

Simple Fact

An optimal prefix-free encoding corresponds to a *full binary tree*

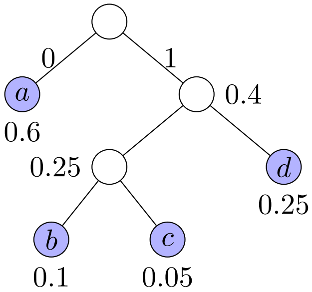
- every node has either zero or two children
- except leaves, each internal node have two children

Proof. If not, there must be a local structure as shown in right picture, we can remove the node with only one child.



An Example

Example. $\langle a, 0.6\rangle, \langle b, 0.05\rangle, \langle c, 0.1\rangle, \langle d, 0.25\rangle$

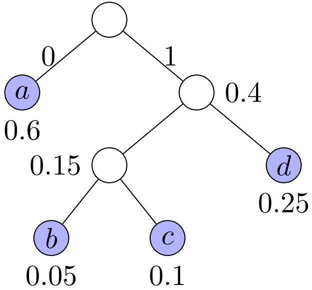


$$(0.05 + 0.1) \times 3 + 0.25 \times 2 + 0.6 \times 1$$
$$= 1.55$$

cost of tree : $L(T)$

depth of i-th symbol in the tree : d_i

$$L(T) = \sum_{i}^{n} f_i \cdot d_i = \sum_{i}^{n} f_i \cdot \ell_i = L(\Gamma)$$



$$(0.05 + 0.1) + 0.15 + 0.25 + 0.4 + 0.6$$
$$= 1.55$$

Define frequency of *internal* node to the sum of its two children

The cost of a tree is the sum of the frequencies of all leaves and internal nodes, except the root.

- for full binary tree with $n > 1$ leaves, there are one root node, $n - 2$ internal nodes

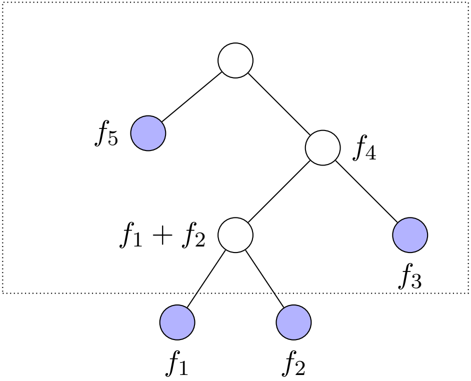
Another way to write the cost function:

$$L(T) = \sum_{i}^{2n-2} f_i$$

Greedy Algorithm

Constructing the full binary tree in a greedy manner:

1. find the two symbols with the smallest frequencies, say 1 and 2
2. make them children of a new node, which then has frequency $f_i + f_j$
3. pull f_1 and f_2 off the list of frequencies, insert $(f_1 + f_2)$, and loop.



Huffman Coding

[David A. Huffman, 1952] A Method for the Construction of Minimum-Redundancy Codes.

- Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.

Huffman Coding

[David A. Huffman, 1952] A Method for the Construction of Minimum-Redundancy Codes.

- Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.

Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. (not trivial)

Huffman Coding

[David A. Huffman, 1952] A Method for the Construction of Minimum-Redundancy Codes.

- Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.

Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. (not trivial)

Shannon's source coding theorem

- the entropy is a measure of the smallest codeword length that is theoretically possible

$$\tilde{\mathrm{H}}(\Gamma) = \sum_{i=1}^{n} p_i \log \frac{1}{p_i}$$

Huffman Coding

[David A. Huffman, 1952] A Method for the Construction of Minimum-Redundancy Codes.

- Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.

Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. (not trivial)

Shannon's source coding theorem

- the entropy is a measure of the smallest codeword length that is theoretically possible

$$\tilde{\mathrm{H}}(\Gamma) = \sum_{i=1}^{n} p_i \log \frac{1}{p_i}$$

Huffman coding is very close to the theoretical limit established by Shannon.

History of Huffman Coding

Huffman coding remains in wide use because of its simplicity, high speed, and lack of patent coverage.

- often used as a "back-end" to other compression methods.
- DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes

History of Huffman Coding

Huffman coding remains in wide use because of its simplicity, high speed, and lack of patent coverage.

- often used as a "back-end" to other compression methods.
- DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes

> In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.
> In doing so, Huffman outdid Fano, who had worked with information theory inventor Claude Shannon to develop a similar code. Building the tree from the bottom up guaranteed optimality, unlike top-down Shannon-Fano coding.

Pseudocode of Huffman Coding

Algorithm 1: HuffmanEncoding($S = \{x_i\}, 0 \leq f(x_i) \leq 1$)

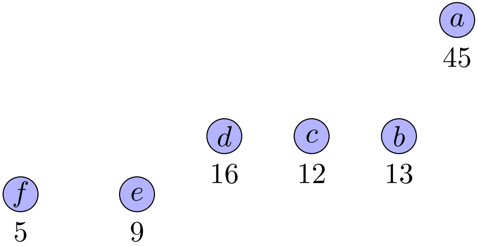
Output: An encoding tree with n leaves

let Q be a priority queue of integers (symbol index), ordered by frequency;
for $i = 1$ *to* n **do** insert(Q, i);
for $k = n + 1$ *to* $2n - 1$ **do**
    $i =$ deletemin(Q), $j =$ deletemin(Q);
    create a node numbered k with children i, j;
    $f(k) \leftarrow f(i) + f(j)$ //i is left child, j is right child;
    insert(Q, k);
end
return Q;

- After each operation, the length of queue decreases by 1. When there is only one element left, the construction of Huffman tree finishes.

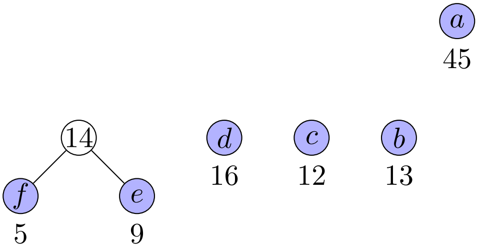
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



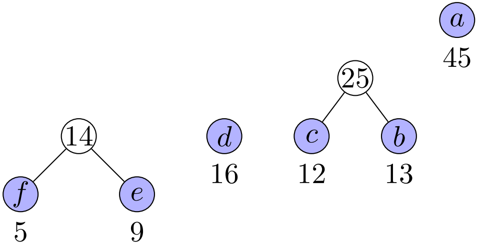
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



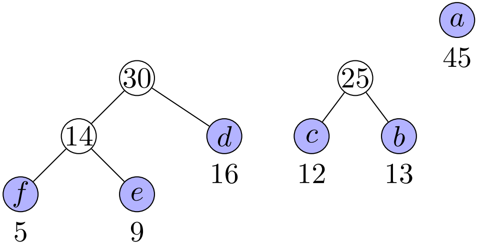
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



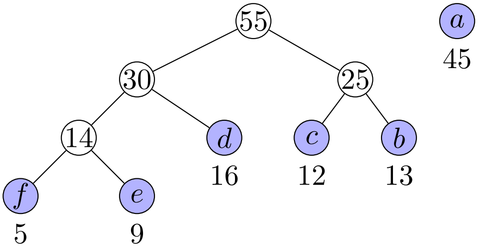
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



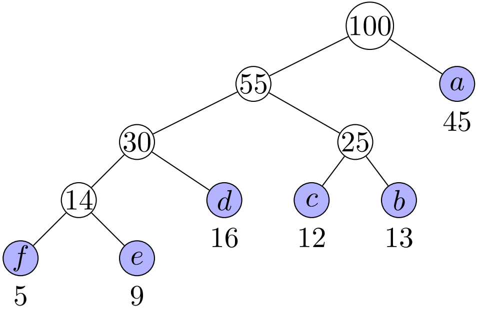
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



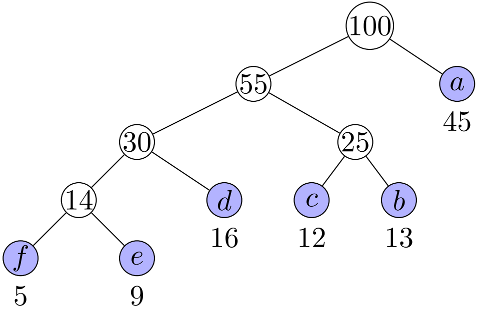
Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



Demo of Huffman Encoding

Input: $\langle a, 0.45\rangle, \langle b, 0.13\rangle, \langle c, 0.12\rangle, \langle d, 0.16\rangle, \langle e, 0.09\rangle, \langle f, 0.05\rangle$



Encoding: $\langle f, 0000\rangle, \langle e, 0001\rangle, \langle d, 001\rangle, \langle c, 010\rangle, \langle b, 011\rangle, \langle a, 1\rangle$

Average code length:
$4 \times (0.05 + 0.09) + 3 \times (0.16 + 0.12 + 0.13) + 1 \times 0.45 = 2.24$

Property of Optimal Prefix-free Encoding: Lemma 1

Lemma 1. Let x and y be two symbols in Γ with smallest frequency, then there exists optimal prefix-free encoding such that the codewords of x and y are equal but only differ in the last bit.

Property of Optimal Prefix-free Encoding: Lemma 1

Lemma 1. Let x and y be two symbols in Γ with smallest frequency, then there exists optimal prefix-free encoding such that the codewords of x and y are equal but only differ in the last bit.

Proof sketch

- Breaking the lemma into cases depending on $|\Gamma|$.
- By the correspondence between encoding scheme and tree, just need to prove the tree T stated by lemma is optimal than trees T' of other forms.

Property of Optimal Prefix-free Encoding: Lemma 1

Lemma 1. Let x and y be two symbols in Γ with smallest frequency, then there exists optimal prefix-free encoding such that the codewords of x and y are equal but only differ in the last bit.

Proof sketch

- Breaking the lemma into cases depending on $|\Gamma|$.
- By the correspondence between encoding scheme and tree, just need to prove the tree T stated by lemma is optimal than trees T' of other forms.

Case $|\Gamma| = 1$, the lemma obviously holds. Only one possibility: one root node.

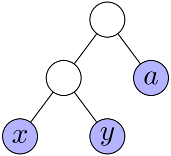
Case $|\Gamma| = 2$, the lemma obviously holds. Only one possibility: one-level full binary tree with two leaves.

Proof of Lemma 1 (continue)

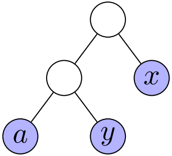
Case $|\Gamma| = 3$, two possibilities T and T'

- T: both x, y are sibling leaves in the second level (by algorithm)
- T': only one of x and y is leaf node in the second level

T



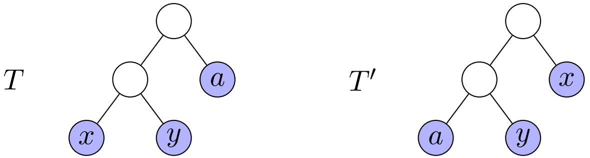
T'



Proof of Lemma 1 (continue)

Case $|\Gamma| = 3$, two possibilities T and T'

- T: both x, y are sibling leaves in the second level (by algorithm)
- T': only one of x and y is leaf node in the second level

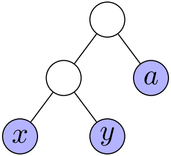


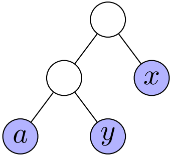
$$L(T') - L(T) = f_a \times 2 + f_x - (f_x \times 2 + f_a) = f_a - f_x \geq 0$$

Proof of Lemma 1 (continue)

Case $|\Gamma| = 3$, two possibilities T and T'

- T: both x, y are sibling leaves in the second level (by algorithm)
- T': only one of x and y is leaf node in the second level

T 

T' 

$$L(T') - L(T) = f_a \times 2 + f_x - (f_x \times 2 + f_a) = f_a - f_x \geq 0$$

$$\Rightarrow T \text{ is optimal}$$

Proof of Lemma 1 (continue)

Case $|\Gamma| \geq 4$, consider the following possible sub-cases:

- both x and y are not sibling leaf nodes in the deepest level: use (x, y) to replace sibling leaf nodes (a, b) in the deepest level (such (a, b) must exist) to obtain T

$$
\begin{aligned}
&L(T') - L(T) = \\
=&f_a d_a + f_b d_b + f_x d_x + f_y d_y - (f_x d_a + f_y d_b + f_a d_x + f_b d_y) \\
=&(d_a - d_x)(f_a - f_x) + (d_b - d_y)(f_b - f_y) \geq 0
\end{aligned}
$$

- x and y are in the deepest level but not sibling nodes: simple swap to obtain T':

$$L(T) = L(T')$$

- only one of x and y in the deepest level, w.l.o.g. assume x is in the deepest level and its sibling is a, swap y and a to obtain T. Similar to $|\Gamma| = 3$, we have:

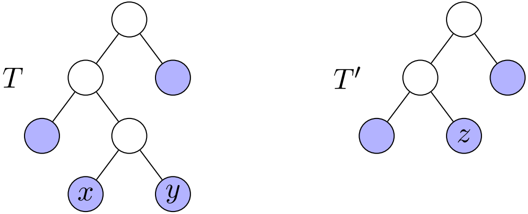
$$L(T') - L(T) \geq 0$$

Properties of Optimal Prefix-free Encoding: Lemma 2

Lemma 2. Let T be the prefix-free encoding binary tree for (Γ, F), x and y be two sibling leaf nodes and z be their parent node. Let T' be the tree for $\Gamma' = (\Gamma - \{x, y\}) \cup \{z\}$ derived from T, where $f'_c = f_c$ for all $c \neq \Gamma - \{x, y\}$, $f_z = f_x + f_y$. Then:

T' is a full binary tree

$$L(T) = L(T') + f_x + f_y$$



Proof of Lemma 2

$\forall c \in \Gamma - \{x, y\}$, we have $d_c = d'_c \Rightarrow$

$$\begin{cases} f_c d_c = f_c d'_c \\ d_x = d_y = d'_z + 1 \\ \Gamma - \{x, y\} = \Gamma' - \{z\} \end{cases}$$

$$\begin{aligned} L(T) =& \sum_{c \in \Gamma} f_c d_c = \left(\sum_{c \in \Gamma - \{x,y\}} f_c d_c \right) + (f_x d_x + f_y d_y) \\ =& \left(\sum_{c \in \Gamma' - \{z\}} f_c d'_c \right) + f_z d'_z + (f_x + f_y) \\ =& L(T') + f_x + f_y \end{aligned}$$

Correctness Proof of Huffman Encoding

Theorem. Huffman algorithm yields optimal prefix-free encoding binary tree for all $|\Gamma| \geq 2$.

Proof. Mathematic induction

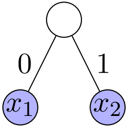
Induction basis. For $|\Gamma| = 2$, Huffman algorithm yields optimal prefix-free encoding.

Induction step. Assume Huffman algorithm encoding yields optimal prefix-free encoding for size k, then it also yields optimal prefix-free encoding for size $k + 1$.

Induction Basis

$k = 2$, $\Gamma = \{x_1, x_2\}$

Any codeword at least require at least one bit. Huffman algorithm yields code word 0 and 1, which is optimal prefix-free encoding.



Induction Step (1/3)

Assume Huffman algorithm yields optimal prefix-free encoding for input size k. Now, consider input size $k+1$, a.k.a. $|\Gamma| = k+1$

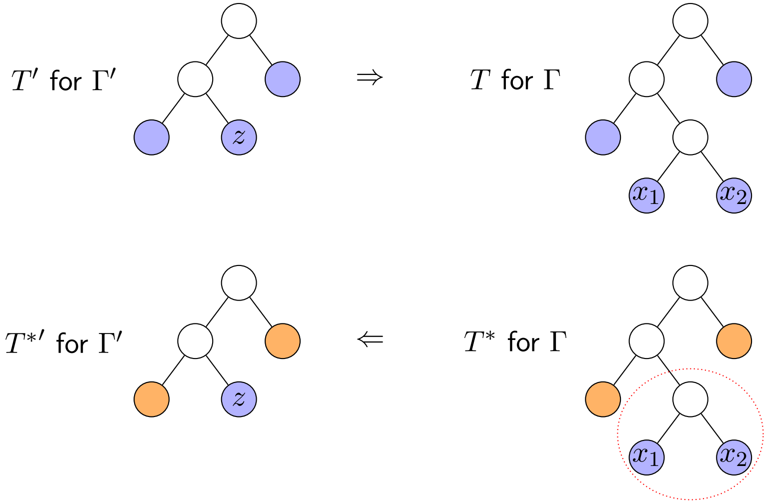
$$\Gamma = \{x_1, x_2, \ldots, x_{k+1}\}$$

Let $\Gamma' = (\Gamma - \{x_1, x_2\}) \cup \{z\}$, $f_z = f_{x_1} + f_{x_2}$

Induction premise $\Rightarrow$ Huffman algorithm generates an optimal prefix-free encoding tree T' for Γ', where frequencies are f_z and f_{x_i} $(i = 3, 4, \ldots, k+1)$.

Induction Step (2/3)

Claim. Append (x_1, x_2) as z's children to T', obtaining T (greedy algorithm's output), which is an optimal prefix-free encoding tree for $\Gamma = (\Gamma' - \{z\}) \cup \{x_1, x_2\}$.



Induction Step (3/3)

Proof. If not, then there exists an optimal prefix-encoding free tree T^*, $L(T^*) < L(T)$.

- Lemma 1 $\Rightarrow$ x and y must be the sibling leaves in the deepest level

Idea. Reduce to the optimality of T' for Γ'

Remove x and y from T^*, obtaining a new encoding tree $T^{*\prime}$ for C'. Lemma 2 $\Rightarrow$

$$\begin{aligned} L(T^{*\prime}) =& L(T^*) - (f_{x_1} + f_{x_2}) \\ <& L(T) - (f_{x_1} + f_{x_2}) \\ =& L(T') \end{aligned}$$

This contradicts to the premise that T' is an optimal prefix-free encoding tree for Γ'.

Application: Files Merge

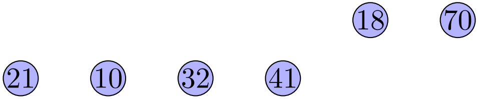
Problem. Given a collection of files $\Gamma = \{1, \ldots, n\}$, In each file, the items have been sorted, f_i denotes the number of items in file i. Now, the task is using 2-way merging sort to merge these files into a single files whose items are sorted.

Represent the merging the process as a bottom-up binary tree.

- leaf nodes: files labeled with $\{1, \ldots, n\}$
- costs of merging file of i and j: parent node of i and j

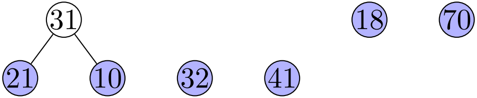
Demo of 2-ary Sequential Merge

Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



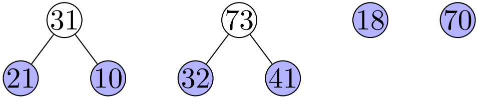
Demo of 2-ary Sequential Merge

Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



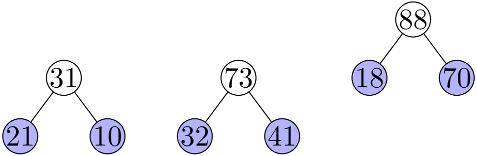
Demo of 2-ary Sequential Merge

Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



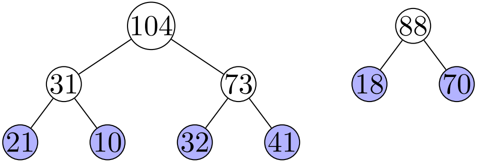
Demo of 2-ary Sequential Merge

Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



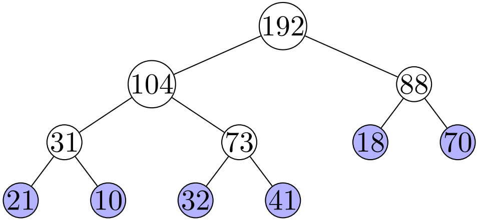
Demo of 2-ary Sequential Merge

Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



Demo of 2-ary Sequential Merge

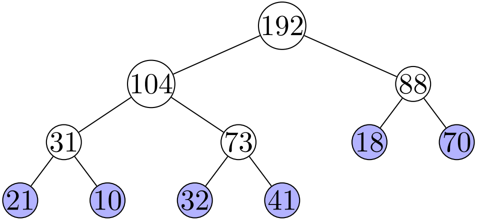
Example. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



Complexity of 2-way Merge (1/2)

Worst-case complexity of merging ordered $A[k]$ and $B[l]$ into ordered $C[k+l]$

- same to merge operation in MergeSort
- $W(k,l) \leq k+l-1$



Bottom-up calculation of worst merging complexity:

$$(21+10-1)+(32+41-1)+(18+70-1)$$
$$+(31+73-1)+(104+88-1)=483$$

Complexity of 2-way Merge (2/2)

Calculation from n leaf nodes

$$(21 + 10 + 32 + 41) \times 3 + (18 + 70) \times 2 - 5 = 483$$

Worst-case complexity

$$W(n) = \left(\sum_{i=1}^{n} f_i d_i \right) - (n - 1)$$

Complexity of 2-way Merge (2/2)

Calculation from n leaf nodes

$$(21 + 10 + 32 + 41) \times 3 + (18 + 70) \times 2 - 5 = 483$$

Worst-case complexity

$$W(n) = \left(\sum_{i=1}^{n} f_i d_i \right) - (n - 1)$$

How to prove the correctness of formula?

Complexity of 2-way Merge (2/2)

Calculation from n leaf nodes

$$(21 + 10 + 32 + 41) \times 3 + (18 + 70) \times 2 - 5 = 483$$

Worst-case complexity

$$W(n) = \left(\sum_{i=1}^{n} f_i d_i \right) - (n - 1)$$

How to prove the correctness of formula?

- The tree is generated in bottom-up manner, and must be a full binary tree. Each non-leaf node corresponds to a merge operation, contribution to $W(n)$ is -1.
- The number of internal nodes $= m$. Except leaf nodes, for all internal nodes: in-degree $= 1$, out-degree $= 2$.

$$\sum \text{out-degree} = 2m, \sum \text{in-degree} = n + (m - 1) \Rightarrow m = n - 1$$

Optimal File Merge

Goal. Find a sequence to minimize $W(n)$

- The problem is in spirit the same of Huffman encoding (except a fixed constant $n-1$).

Solution. Treat the item numbers as frequency, apply Huffman algorithm to generate the merging tree.

Special example. $n = 2^k$ files and each file has the same number of items. In this case, Huffman algorithm generate a prefect binary tree, which is same as iterated version 2-way merge sort.

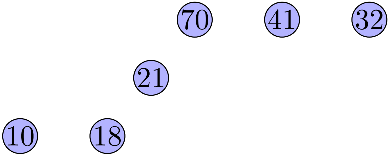
- This also demonstrates the optimality of MergeSort.

Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$

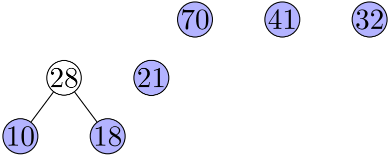
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



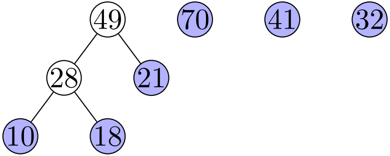
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



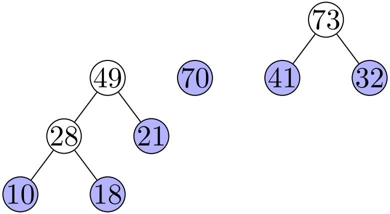
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



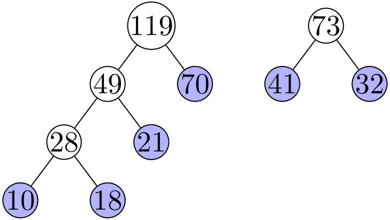
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



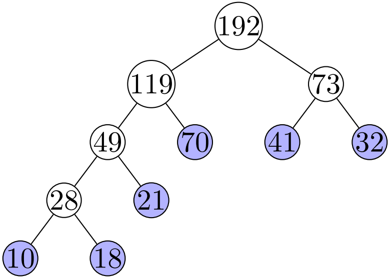
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



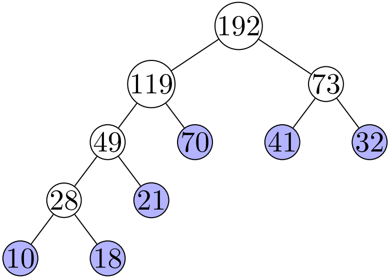
Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



Demo of Huffman Tree Merging

Input. $\Gamma = \{21, 10, 32, 41, 18, 70\}$



Cost. $(10 + 18) \times 4 + 21 \times 3 + (70 + 41 + 32) \times 2 - 5 = 456 < 483$

Recap of Huffman Coding

Refine the problem as optimal prefix-free encoding

Model the encoding scheme as building an optimized full binary tree

- Optimize function: the cost of tree

Prove the greedy construction is optimal

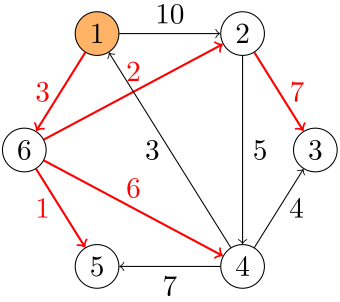
- Lemma 1: prove the optimal encoding tree must satisfy certain local structure
- Lemma 2: prove the optimality by induction on the size of Γ
  - Optimal for $k \Rightarrow$ Optimal for $k+1$
  - The local structure is useful for arguing optimality

Outline

1. Huffman Coding
2. Single Source Shortest Path Problem
   - Dijkstra's Algorithm
3. Minimal Spanning Tree
   - Kruskal's Algorithm
   - Prim's Algorithm

Single Source Shortest Path (SSSP) Problem

Problem. Given a directed graph $G = (V, E)$, edge weight $e(i,j) \geq 0$, calculate the shortest path from a source node $s \in V$ to all the nodes inside G.



$$d(1,2) = 5 : \langle 1 \to 6 \to 2 \rangle$$

$$d(1,3) = 12 : \langle 1 \to 6 \to 2 \to 3 \rangle$$

$$d(1,4) = 9 : \langle 1 \to 6 \to 4 \rangle$$

$$d(1,5) = 4 : \langle 1 \to 6 \to 5 \rangle$$

$$d(1,6) = 3 : \langle 1 \to 6 \rangle$$

Applications

Wide applications in the real world

- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Outline

1. Huffman Coding
2. Single Source Shortest Path Problem
   - Dijkstra's Algorithm
3. Minimal Spanning Tree
   - Kruskal's Algorithm
   - Prim's Algorithm

Dijkstra was a Dutch systems scientist, programmer, software engineer, science essayist, and pioneer in computing science.

- Algorithmic work
- Compiler construction and programming language research,
- Programming paradigm and methodology
- Operating system research
- Concurrent computing and programming
- Distributed computing
- Formal specification and verification

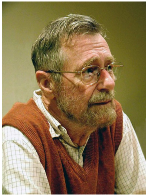


Figure: Edsger Wybe Dijkstra

Dijkstra's Style

No reference. Dijkstra chose this way of working to preserve his self-reliance.

His approach to teaching was unconventional

- A quick and deep thinker while engaged in the act of lecturing
- Never followed a textbook, with the possible exception of his own while it was under preparation.
- Assigned challenging homework problems, and would study his students' solutions thoroughly.
- Conducted his final examinations orally, over a whole week. Each student was examined in Dijkstra's office or home, and an exam lasted several hours.

Dijkstra's Algorithm

Intuition. (Breadth-First Search) Explore the unknown world step by step, the cognition on each step is correct.

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined the shortest path distance from s to u, as well as a set of unexplored nodes U.

1. Initialize $S = \emptyset$, $d(s, s) = 0$, $d(s, v) = \infty$; $U = V$
2. Repeatedly choose unexplored node $u \in U$ with

$$\min_{u \in U} d(s, u)$$

   - add u to S
   - for the rest nodes $v \in U$, update $d(s, v) = d(s, u) + e(u, v)$ if the right part is shorter and set prev$(v) = u$
3. Finishes when $S = V$, a.k.a. $U = \emptyset$.

Implement Details

Data structure:

- S: nodes have been explored
- U: nodes have not been explored
- $d(s, v)$: the length of current shortest path from s to v. If $v \in S$, then it is the final length of shortest path.
- prev(v): preceding node of v, used to track path

Critical optimization. Priority queue $\Rightarrow$ computing $\min_{u \in U} d(s, u)$

- For each $u \notin S$, $d(s, u)$ can only decrease (because S only increases). Suppose u is added to S and there is an edge (u, v) from u to $v \in U$. Then, it suffices to update:

$$d(s, v) = \min\{d(s, v), d(s, u) + e(u, v)\}$$

Priority Queue

Priority queue (usually implemented via a heap). It maintains a set of elements with associated numeric key values and supports the following operations.

- Insert. Add a new element to the set.
- Decrease-key. Accommodate the decrease in key value of a particular element.
- Delete-min. Return the element with smallest key, and remove it from the set.
- Make-queue. Build a priority queue out of the given elements, with the given key values.

Dijkstra's Algorithm to SSSP

Algorithm 2: Dijkstra$(G = (V, E), s)$

$S = \{\perp\}$, $d(s, s) = 0$, $U = V$;
for $u \in U - \{s\}$ **do** $d(s, u) = \infty$, prev$(u) = \perp$;
$Q \leftarrow$ makequeue(V) //using dist-values as keys;
while $S \neq V$ **do**
    $u =$ deletemin(Q), $S = S \cup u$;
    for *all* $v \in U$ **do** //update
        if $d(s, v) > d(s, u) + e(u, v)$ **then**
          $d(s, v) = d(s, u) + e(u, v)$;
        prev$(v) = u$; decreasekey(Q, v) ;
    end
end

- Dijkstra's algorithm works correctly for both directed and undirected graphs provided that there is no negative edges.

Dijkstra's algorithm: Which priority queue?

Cost of priority queue operations. $|V|$ insert, $|V|$ delete-min, $O(|E|)$ decrease-key (think why)

Performance: depends heavily on priority queue implementation

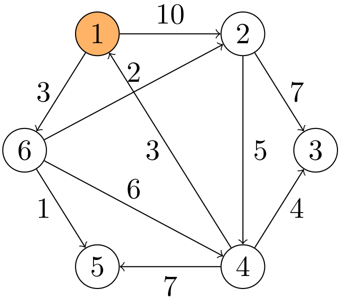
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci/Brodal best in theory, but not worth implementing.

Implementation	insert	delete-min	decrease-key	total cost
unordered array	$O(1)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\|E\| \log \|V\|)$
d-ary heap	$O(d \log_d \|V\|)$	$O(d \log_d \|V\|)$	$O(\log_d \|V\|)$	$O(\|E\| \log_{\|E\|/\|V\|} \|V\|)$
Fibonacci heap	$O(1)$	$O(\log \|V\|)^\dagger$	$O(1)^\dagger$	$O(\|E\| + \|V\| \log \|V\|)$
Brodal queue	$O(1)$	$O(\log \|V\|)$	$O(1)$	$O(\|E\| + \|V\| \log \|V\|)$

† amortized

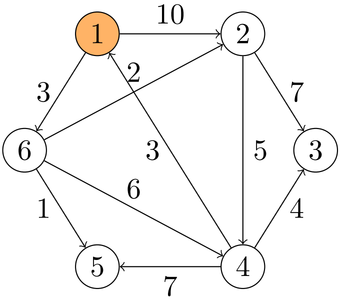
Demo of Dijkstra's Algorithm

Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



Demo of Dijkstra's Algorithm

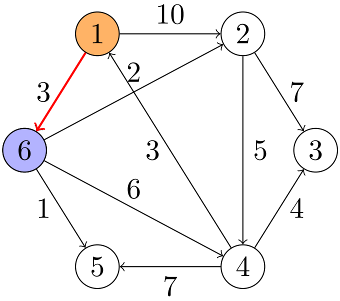
Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$$S = \{1\}$$
$$d(1,1) = 0$$
$$d(1,2) = 10$$
$$d(1,6) = 3$$
$$d(1,3) = \infty$$
$$d(1,4) = \infty$$
$$d(1,5) = \infty$$

Demo of Dijkstra's Algorithm

Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$S = \{1, 6\}$

$d(1, 1) = 0$

$d(1, 6) = 3$

$d(1, 2) = 5$

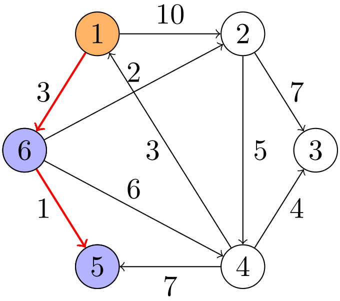
$d(1, 4) = 9$

$d(1, 5) = 4$

$d(1, 3) = \infty$

Demo of Dijkstra's Algorithm

Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$S = \{1, 6, 5\}$

$d(1, 1) = 0$

$d(1, 6) = 3$

$d(1, 5) = 4$

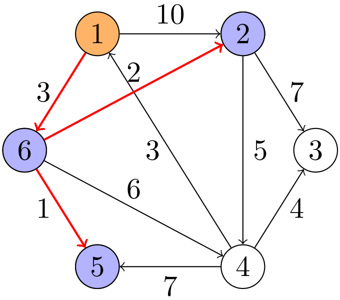
$d(1, 2) = 5$

$d(1, 4) = 9$

$d(1, 3) = \infty$

Demo of Dijkstra's Algorithm

Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$S = \{1, 6, 5, 2\}$

$d(1, 1) = 0$

$d(1, 6) = 3$

$d(1, 5) = 4$

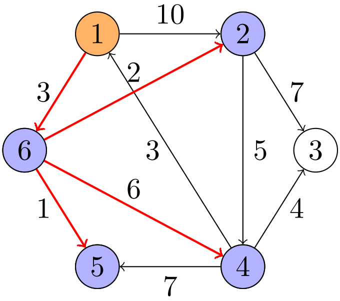
$d(1, 2) = 5$

$d(1, 4) = 9$

$d(1, 3) = 12$

Demo of Dijkstra's Algorithm

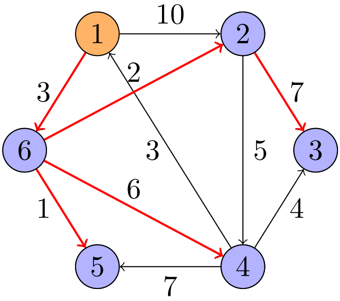
Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$$S = \{1, 6, 5, 2, 4\}$$
$$d(1,1) = 0$$
$$d(1,6) = 3$$
$$d(1,5) = 4$$
$$d(1,2) = 5$$
$$d(1,4) = 9$$
$$d(1,5) = 12$$

Demo of Dijkstra's Algorithm

Input. $G = (V, E)$, $s = 1$, $V = \{1, 2, 3, 4, 5, 6\}$



$S = \{1, 6, 5, 2, 4, 3\}$

$d(1, 1) = 0$

$d(1, 6) = 3$

$d(1, 5) = 4$

$d(1, 2) = 5$

$d(1, 4) = 9$

$d(1, 3) = 12$

Proof of Correctness: Dijkskra's Algorithm (1/2)

Proposition. For each node $u \in S$, $d(s, u)$ is the length of the shortest $s \rightsquigarrow u$ path. a.k.a. Dijkstra's k-th step result is also the final result.

Proof. By induction on $|S|$.

Induction basis. $|S| = 1$, $S = \{s\}$, $d(s, s) = 0$. Obviously holds.

Induction step. Assume Proposition is true for $|S| = k \geq 1$. Prove the proposition also holds for $|S| = k + 1$.

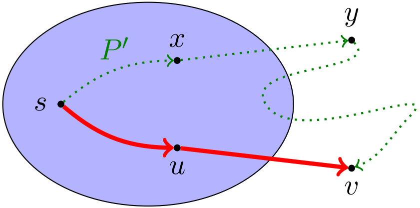
Proof of Correctness: Dijkskra's Algorithm (2/2)

Let v be the next node ($k+1$ step) added to S, and let (u,v) be the final edge, thus $d(s,v) = d(s,u) + e(u,v)$

Consider any $s \rightsquigarrow v$ path P. We show that $\ell(P) \geq d(s,v)$.

- Let (x,y) be the first edge in P that leaves S, and let P' be the subpath to x, then P is already too long as soon as it reaches y. (x may equal u; y may equal v)

$$
\begin{aligned}
\ell(P) &\geq \ell(P') + e(x,y) + \ell(y,v) && \text{//definition of } P \\
&\geq \underline{d(s,x)} + e(x,y) + 0 && \text{//induction hypothesis} \\
&\geq d(s,y) && \text{//definition of } d \\
&\geq d(s,v) && \text{//Dijkstra chose } v \text{ instead of } y
\end{aligned}
$$



Extensions of Dijkstra's algorithm

Dijkstra's algorithm and proof extend to several related problems:

- Shortest paths in undirected graphs
- Maximum capacity paths
- Maximum reliability paths

Outline

1 Huffman Coding

2 Single Source Shortest Path Problem
- Dijkstra's Algorithm

3 Minimal Spanning Tree
- Kruskal's Algorithm
- Prim's Algorithm

Motivation of MST

Real world problem. You are asked to network a collection of computers by linking them. Each link has a maintenance cost.

What is the cheapest possible network?

Motivation of MST

Real world problem. You are asked to network a collection of computers by linking them. Each link has a maintenance cost.

What is the cheapest possible network?

This translates into a graph problem:

- nodes: computers
- undirected edges: potential links
- edge's weight: maintenance cost

Optimization goal. Pick enough edges so that all the nodes are connected and the total weight is minimal.

What are the properties of these edges?

One immediate observation. Optimal set of edges cannot contain a cycle, since removing an edge from this cycle would reduce the cost without compromising connectivity.

Fact 1. Removing a cycle edge cannot disconnect an undirected graph.

What are the properties of these edges?

One immediate observation. Optimal set of edges cannot contain a cycle, since removing an edge from this cycle would reduce the cost without compromising connectivity.

Fact 1. Removing a cycle edge cannot disconnect an undirected graph.

The solution must be connected and acyclic

- undirected graphs of this kind are called *trees*
- the particular tree is the one with minimum total weight, known as *minimal spanning tree*.

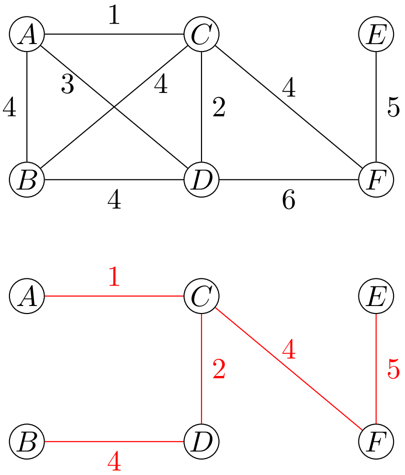
Minimal Spanning Tree

Spanning tree. Let $G = (V, E)$ be an undirected graph, where w_e is the weight of edge e.

- A connected and acyclic subgraph $T = (V, E')$ is called a spanning tree of G.
- The weight of the tree T is weight$(T) = \sum_{e \in E'} w_e$
- The minimal spanning tree is the tree that minimizes weight(T)

The minimal spanning tree may not be unique.

Example of Minimal Spanning Tree



Can you spot another?

Properties of Trees

Definition of Tree. An undirected graph that is connected and acyclic.

Simplicity of structure make the notion of tree extremely useful

Fact of Tree (relation between nodes and edges)

Fact. A tree with n nodes has $n-1$ edges.

Starting from an empty graph and building the tree one edge at a time. Initially the n nodes are disconnected from one another.

- First edge: connect two nodes ($n-2$ nodes are left)
- Then: an edge adds in, a node is connected

Total edges: $1+(n-2)=n-1$

Viewing initially disconnected n nodes as n separate components

- When a particular edge $\langle u, v\rangle$ comes up, it merges two components that u and v previously lie in.
- Adding the edge then merges these two components, reducing the total number of connected components by 1.
- Over the course of this incremental process, the number of connected components decreases from n to $1 \Rightarrow n-1$ edges must have been added along the way

Property 1 of Tree

Property 1. Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Property 1 of Tree

Property 1. Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Proof idea. Using definition, just need to show G is acyclic.

Suppose it is cyclic, we can run the following iterative procedure to make it acyclic

1. remove one edge from a cycle each time
2. terminates with some graph $G' = (V, E')$, $E' \subseteq E$, which is acyclic.

Operation $\Rightarrow$ G' is still connected $\Rightarrow$ G' is a tree (by def of tree)

Fact $\Rightarrow |E'| = |V| - 1 \Rightarrow E' = E \Rightarrow G' = G$

- No edges were removed, and G was acyclic to start with.

Property 1 of Tree

Property 1. Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Proof idea. Using definition, just need to show G is acyclic.
Suppose it is cyclic, we can run the following iterative procedure to make it acyclic

1. remove one edge from a cycle each time
2. terminates with some graph $G' = (V, E')$, $E' \subseteq E$, which is acyclic.

Operation $\Rightarrow$ G' is still connected $\Rightarrow$ G' is a tree (by def of tree)

Fact $\Rightarrow |E'| = |V| - 1 \Rightarrow E' = E \Rightarrow G' = G$

- No edges were removed, and G was acyclic to start with.

We can tell whether a connected graph is a tree just by counting how many edges it has.

Property 2 of Tree (Another Characterization)

Property 2. An undirected graph $G = (V, E)$ is a tree if and only if there is a unique path between any pair of nodes.

Property 2 of Tree (Another Characterization)

Property 2. An undirected graph $G = (V, E)$ is a tree if and only if there is a unique path between any pair of nodes.

Forward direction

- In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

Property 2 of Tree (Another Characterization)

Property 2. An undirected graph $G = (V, E)$ is a tree if and only if there is a unique path between any pair of nodes.

Forward direction

- In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

Backward direction (def: connected+acyclic $\Rightarrow$ tree)

- G has a path between any two nodes $\Rightarrow$ G is connected
- If these paths are unique, then the G is acyclic (since any pair of nodes in a cycle have at least two paths between them).

Brief Summary

The above properties give three criteria to decide if an undirected graph is a tree

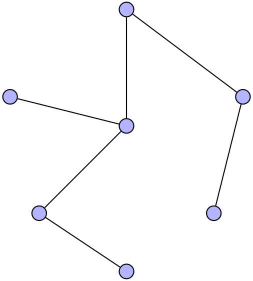
1. Definition: connected and acyclic
2. Property 1: connected and $|V| = |E| - 1$
3. Property 2: there is a unique path between any two nodes

Spanning Tree: Proposition 1

Proposition 1. If T is a spanning tree of G, $e \notin T$, then $T \cup \{e\}$ contains a cycle C.

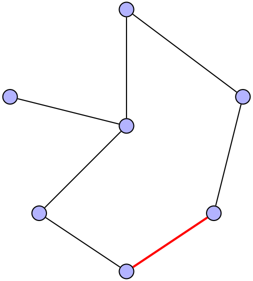
Spanning Tree: Proposition 1

Proposition 1. If T is a spanning tree of G, $e \notin T$, then $T \cup \{e\}$ contains a cycle C.



Spanning Tree: Proposition 1

Proposition 1. If T is a spanning tree of G, $e \notin T$, then $T \cup \{e\}$ contains a cycle C.

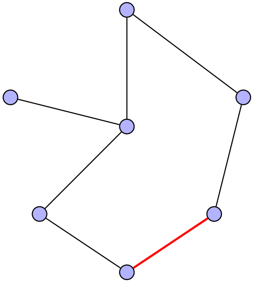


Spanning Tree: Proposition 2

Proposition 2. Removing any edge in the cycle C (in the context of Proposition 1), yields another spanning tree T' of G.

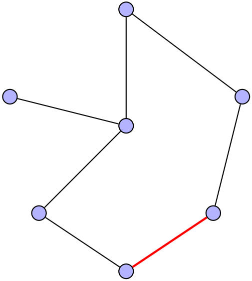
Spanning Tree: Proposition 2

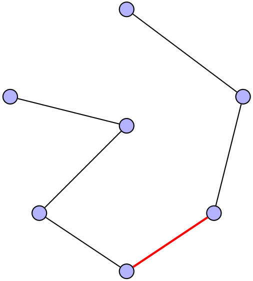
Proposition 2. Removing any edge in the cycle C (in the context of Proposition 1), yields another spanning tree T' of G.



Spanning Tree: Proposition 2

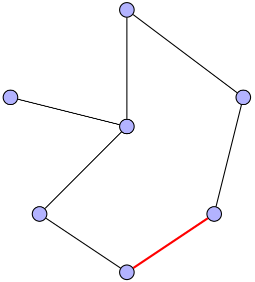
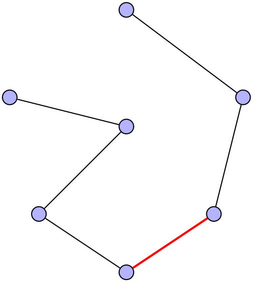
Proposition 2. Removing any edge in the cycle C (in the context of Proposition 1), yields another spanning tree T' of G.





Spanning Tree: Proposition 2

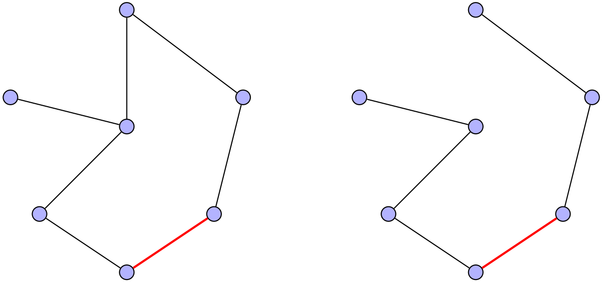
Proposition 2. Removing any edge in the cycle C (in the context of Proposition 1), yields another spanning tree T' of G.

all nodes are still connected + Property 1 $\Rightarrow$ Proposition 2

Spanning Tree: Proposition 2

Proposition 2. Removing any edge in the cycle C (in the context of Proposition 1), yields another spanning tree T' of G.



all nodes are still connected + Property 1 $\Rightarrow$ Proposition 2

This proposition gives a method of creating a new spanning tree from an existing spanning tree.

Applications of MST

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Network design (communication, electrical, computer, road).
- Approximation algorithms for $\mathcal{NP}$-hard problems (e.g., TSP, Steiner tree).

The Cut Property

Say that in the process of building MST, we have already chosen some edges and are so far on the right track.

Which edge should we add next?

If there is a correct strategy, then we can solve MST iteratively. The following lemma gives us a lot of flexibility in our choice.

The Cut Property

Say that in the process of building MST, we have already chosen some edges and are so far on the right track.

Which edge should we add next?

If there is a correct strategy, then we can solve MST iteratively. The following lemma gives us a lot of flexibility in our choice.

A cut of V is a partition of V, say, $(S, V-S)$. A cut is compatible with a set of edges X if no edge in X cross between S and $V-S$.

Cut property. Suppose edges X are part of a MST of $G=(V,E)$. Let $(S, V-S)$ be any cut compatible with X, and e be the lightest edge across the cut. Then, $X \cup \{e\}$ is part of some MST.

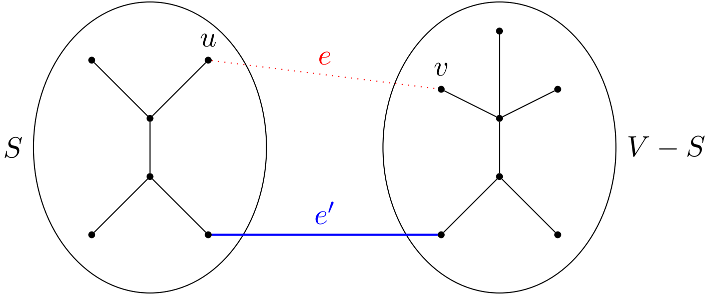
- cut property guarantees that it is always safe to add the lightest edge across any cut, provided it is compatible with X.

Proof of Cut Property

Edges X are part of some MST T (partial solution, on the right track).

- If the new edge e also happens to be part of T, then there is nothing to prove.
- So assume $e \notin T$, we will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.
  1. Add $e = (u, v)$ to T. T is connected, it already has a path between u and v, adding e creates a cycle. This cycle must also have some other edge e' across the cut $(S, V - S)$.
  2. Remove edge e', we are left with $T' = T \cup \{e\} - \{e'\}$:

Proposition 1 + Proposition 2 $\Rightarrow$ T' is still a spanning tree



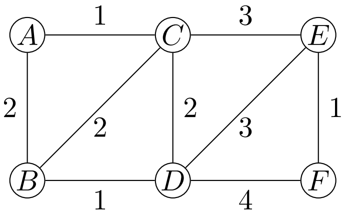
T' is an MST. Proof idea: compare its weight to that of T

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e')$$

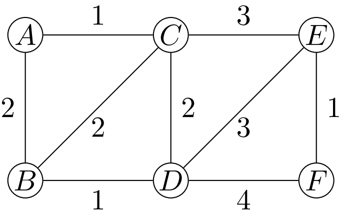
Both e and e' cross between S and $V-S$, and e is the lightest edge of this type.

$$w(e) \le w(e') \Rightarrow \text{weight}(T') \le \text{weight}(T)$$
$$T \text{ is an MST} \Rightarrow \text{weight}(T') = \text{weight}(T) \Rightarrow T' \text{ is also an MST}$$

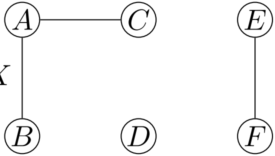
The Cut Property at Work



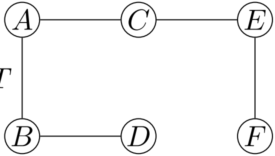
The Cut Property at Work



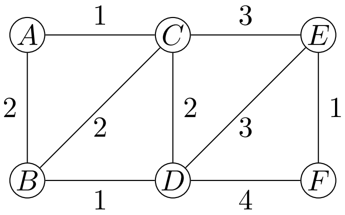
edges X



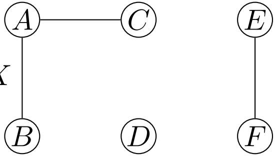
MST T



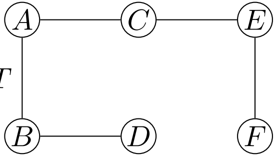
The Cut Property at Work



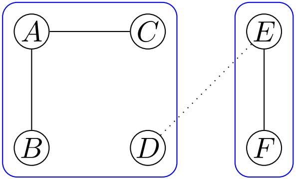
edges X



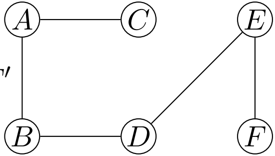
MST T



the cut



MST T'



General MST Algorithm Based on Cut Property

Algorithm 3: GeneralMST(G): output MST defined by X

```
X = ∅ //edges picked so far;
while |X| < |V| − 1 do
    pick a set S ⊂ V for which X has no edges between S
     and V − S //find a cut compatible with X;
    let e ∈ E be the minimal-weight edge between S and
     V − S;
    X ← X ∪ {e};
end
```

Next, we describe two famous MST algorithms following this template.

Outline

1. Huffman Coding

2. Single Source Shortest Path Problem
   - Dijkstra's Algorithm

3. Minimal Spanning Tree
   - Kruskal's Algorithm
   - Prim's Algorithm

Kruskal's Algorithm (edge-by-edge)

Joseph Kruskal [American Mathematical Society, 1956]

Kruskal's Algorithm (edge-by-edge)

Joseph Kruskal [American Mathematical Society, 1956]

Rough idea. Start with the empty graph, then select edges from E according to the following rule

Repeatedly add the next lightest edge that doesn't produce a cycle

Kruskal's Algorithm (edge-by-edge)

Joseph Kruskal [American Mathematical Society, 1956]

Rough idea. Start with the empty graph, then select edges from E according to the following rule

Repeatedly add the next lightest edge that doesn't produce a cycle

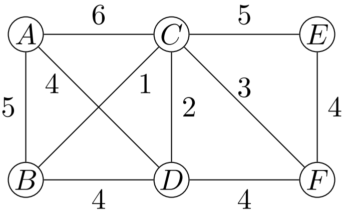
---

Kruskal's algorithm constructs the tree *edge-by-edge*, apart from taking care to avoid cycles, simply picks the cheapest edge at the moment.

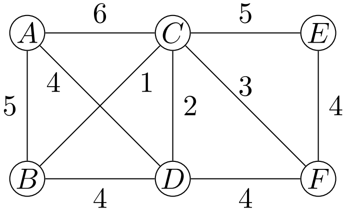
- This is a greedy algorithm: every decision it makes is the one with the most obvious immediate advantage.

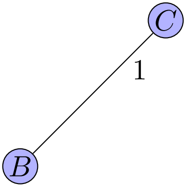
X is initially empty, check every possible cut S and $V - S$.

Demo of Kruskal's Algorithm

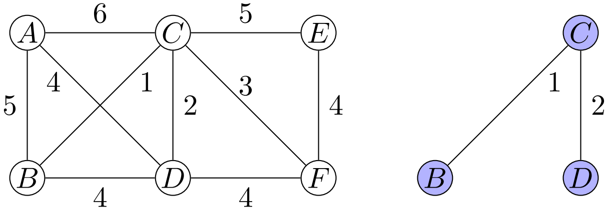


Demo of Kruskal's Algorithm

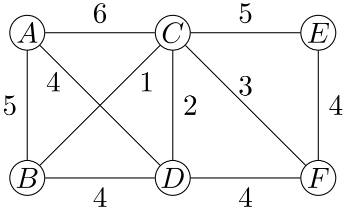


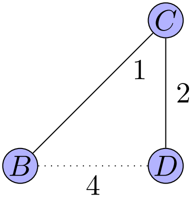


Demo of Kruskal's Algorithm

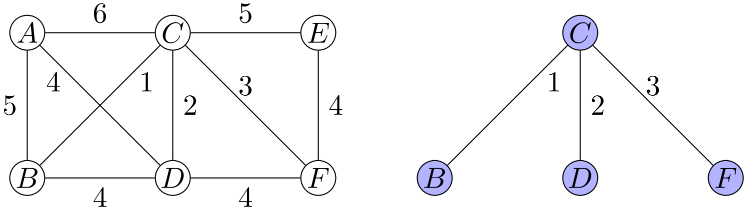


Demo of Kruskal's Algorithm

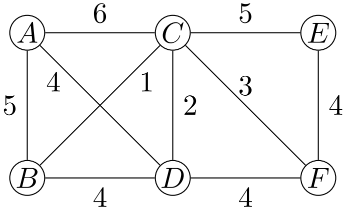


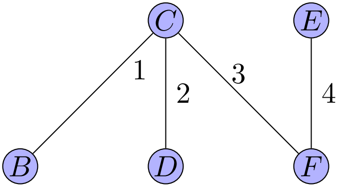


Demo of Kruskal's Algorithm

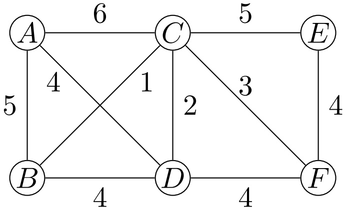


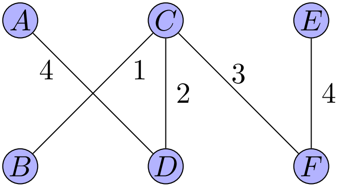
Demo of Kruskal's Algorithm





Demo of Kruskal's Algorithm





Details of Kruskal's Algorithm

At any given moment, the edges it has already chosen form:

- a partial solution X: a collection of connected components, each of which has a tree structure

The next edge e to be added connects two of these components, say, T_1 and T_2

- viewing T_1 as $S \Rightarrow (T_1, V - T_1)$ forms a cut compatible with X
- e is the lightest edge that doesn't produce a cycle, it is certainly to be the lightest edge between T_1 and $V - T_1$

Kruskal's algorithm implicitly searches the lightest crossed edge among all possible compatible cuts

Implementation Details

Select-and-Check. At each stage, the algorithm chooses an edge to add to its current partial solution.

- To do so, it needs to test each candidate edge (u, v) to see whether u and v lie in different components $\Rightarrow$ otherwise the edge produces a cycle

This test implicitly ensures that X is compatible with any cut combination of the form $(T_i, V - T_i)$

Merge. Once an edge is chosen, the corresponding components need to be merged.

What kind a data structure supports such operations?

Data Structure

Model algorithm's state as a collection of *disjoint sets*, each contains the nodes of a particular component.

Initially each node is a component by itself.

- makeset(x): create a singleton set containing just x

Repeatedly test pair of nodes (endpoints of candidate edge) to see if they belong to the same set.

- find(x): to which set does x belong?

Whenever we add an edge, merging two components

- union(x, y): merge the sets containing x and y

Pseudocode of Kruskal's Algorithm

Algorithm 4: Kruskal(G): output MST defined by X

sort the edges E by weight;
for *all* $u \in V$ **do** makeset(u);
$X = \emptyset$ //edge set;
while $|X| < |V| - 1$ **do**
    for *all edges* $(u, v) \in E$ *(in ascending order)* **do**
        if find(u) $\neq$ find(v) **then** //check if X is compatible with the candidate cut
            $X \leftarrow X \cup \{(u, v)\}$, union($u, v$), remove (u, v) from E;
        end
    end
end

Complexity Analysis

- sort E: $O(|E| \log |E|)$
- makeset(u): $O(|V|)$
- find(x): $2|E| \cdot O(\log |V|)$

Outline

1 Huffman Coding

2 Single Source Shortest Path Problem
- Dijkstra's Algorithm

3 Minimal Spanning Tree
- Kruskal's Algorithm
- Prim's Algorithm

Prim's Algorithm (node-by-node)

First discovered by Czech mathematician Vojtěch Jarník 1930, later rediscovered and republished by Robert C. Prim in 1957, and Edsger W. Dijkstra in 1959. Thus, known as the DJP algorithm.

Rough idea

1. Initially: $X = \emptyset$, $S = \{u\}$, u could be an arbitrary node
2. Greedy choice: On each iteration, select the lightest edge $e_{u,v}$ that connects S and $V - S$, where $u \in S$, $v \in V - S$. Add $e_{u,v}$ to T, add v to S.
3. Continue the procedure until $S = V$.

Prim's algorithm is a popular alternative to Kruskal's algorithm, another implementation of the General algorithm

- X is initially empty, S is initially any node
- X always forms a subtree, S is the vertices set of X after first step

We can equivalently think of S as growing to include $v \notin S$ of smallest cost.

$$\text{cost}(v) = \min_{u \in S} w(u, v)$$

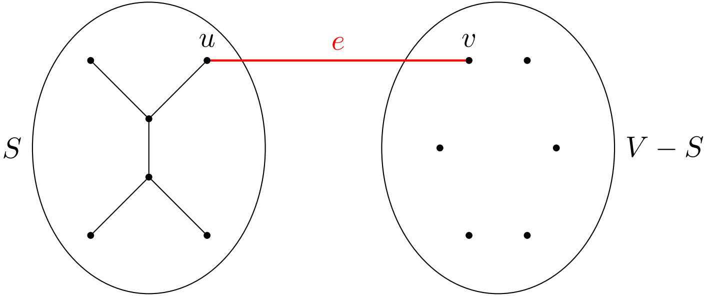
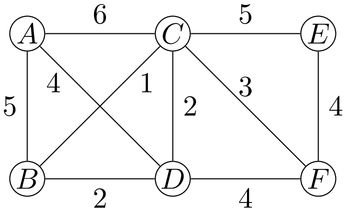


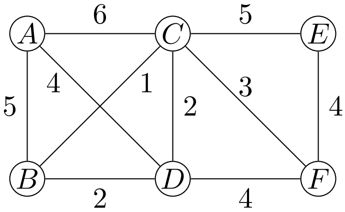
Figure: $T = (S, X)$ forms a tree, and S consists of its vertices.

Demo of Prim Algorithm



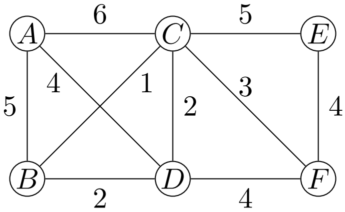
Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

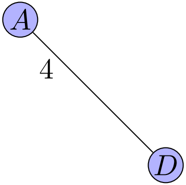
Demo of Prim Algorithm



Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

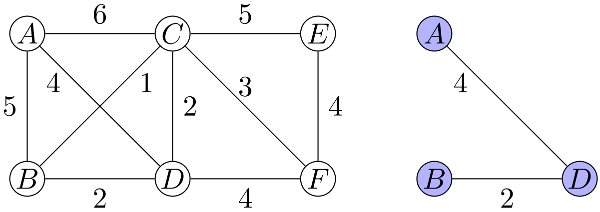
Demo of Prim Algorithm





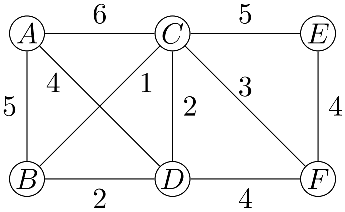
Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

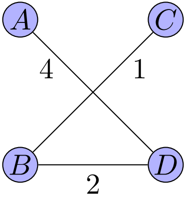
Demo of Prim Algorithm



Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

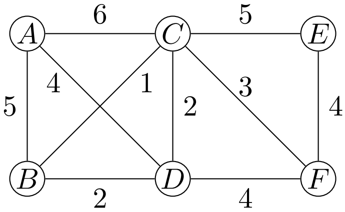
Demo of Prim Algorithm

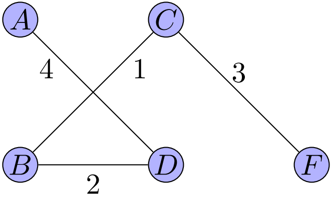




Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

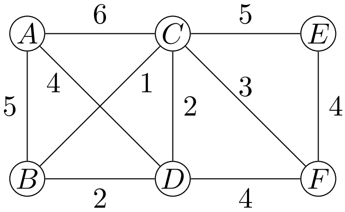
Demo of Prim Algorithm

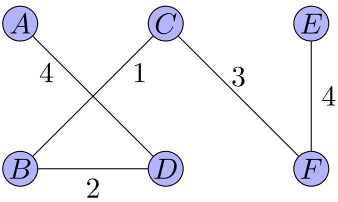




Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Demo of Prim Algorithm





Set S	A	B	C	D	E	F
A		$5/A$	$6/A$	$4/A$	$\infty/\perp$	$\infty/\perp$
A, D		$2/D$	$2/D$		$\infty/\perp$	$4/D$
A, D, B			$1/B$		$\infty/\perp$	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Key Data Structure

At every step, Prim's algorithm has to find the lightest edge that connects S and $V - S$.

How to implement this operation? What kind of data structure could be of help?

We use priority queue.

Pseudocode of Prim Algorithms

Algorithm 5: Prim(G): output MST defined by array prev

for *all* $u \in V$ **do** cost$(u) = \infty$, prev$(u) = \perp$;
pick any initial node u_0: cost$(u_0) = 0$;
$Q =$ makequeue(V);
while Q *is not empty* **do**
    $u =$ deletemin(Q) //current closest point in $V - S$ to S;
    for *each* $v \in V$ **do**
        if $w(u, v) <$ cost(v) **then**
            cost$(v) = w(u, v)$, prev$(v) = u$;
            //update cost-value after adding u to S;
            decreasekey(Q, v)
        end
    end
end

- Q is a priority queue, using cost-value as keys.

Correctness Proof of Prim Algorithm: Mathematical Induction

Proposition. For any $k < n$, there exists a MST containing the edges selected by Prim algorithm in the first k steps.

- Prim algorithm selects one edge in each step, selects $n-1$ edges in total. Thus, the proposition proves the correctness of Prim algorithm.

Proof sketch. Mathematical induction on steps.

Induction basis. $k=1$, there exists a MST T that contains $e_{u,i}$, where $e_{u,i}$ is the minimal-weight edge connected to node u.

Induction step. Assume the edges selected by the first k steps forms a subset of some MST, so does the first $k+1$ steps.

Induction Basis

Claim: There exists a MST T that contains the minimal-weight edge $e_{u,i}$.

Proof. Let T be a MST. If T does not contain $e_{u,i}$, then $T \cup \{e_{u,i}\}$ must contain a cycle, and the cycle has another edge $e_{u,j}$ connecting to node u. Replacing $e_{u,j}$ with $e_{u,i}$ we obtain T^*, T^* is also a spanning tree.

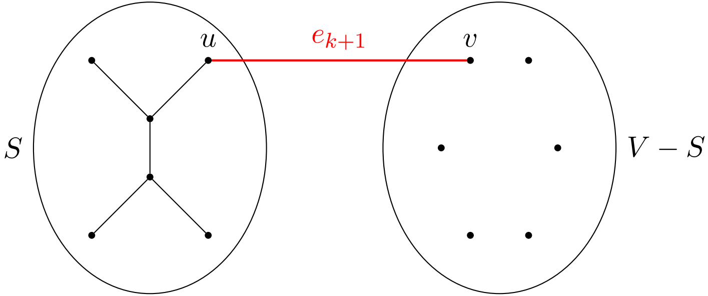
- If $e_{u,i} < e_{u,j}$, then weight$(T^*) <$ weight(T). This contradicts to the hypothesis that T is a MST.
- If $e_{u,i} = e_{u,j}$, then weight$(T^*) =$ weight(T). Then, T^* is a MST that contains $e_{u,i}$.

Induction Steps (1/2)

After k steps, Prim algorithm selects edges $e_1, e_2, \ldots, e_k$. The nodes of these edges form a node set S.

Premise. $\exists$ MST $T = (V, E)$ that contains $(e_1, \ldots, e_k)$.

Let the $k+1$ step choice is $e_{k+1} = (u, v)$, $u \in S$, $v \in V - S$.

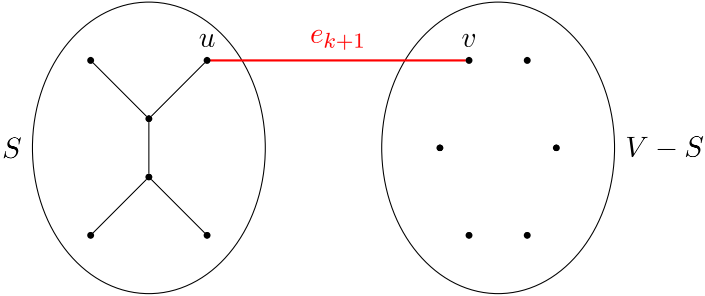


Induction Steps (1/2)

After k steps, Prim algorithm selects edges $e_1, e_2, \ldots, e_k$. The nodes of these edges form a node set S.

Premise. $\exists$ MST $T = (V, E)$ that contains $(e_1, \ldots, e_k)$.

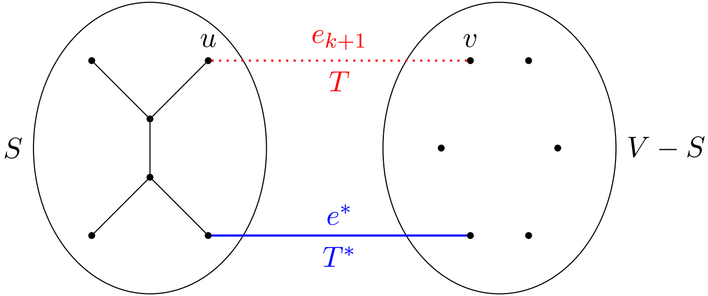
Let the $k+1$ step choice is $e_{k+1} = (u, v)$, $u \in S$, $v \in V - S$.



Case $e_{k+1} \in E$: the induction step of Prim algorithm at $k+1$ step is obviously correct.

Induction Step (2/2)

Case $e_{k+1} \notin E$: adding e_{k+1} to E would create a cycle between (u, v). In this cycle, $\exists$ another edge e^* connecting S and $V - S$.

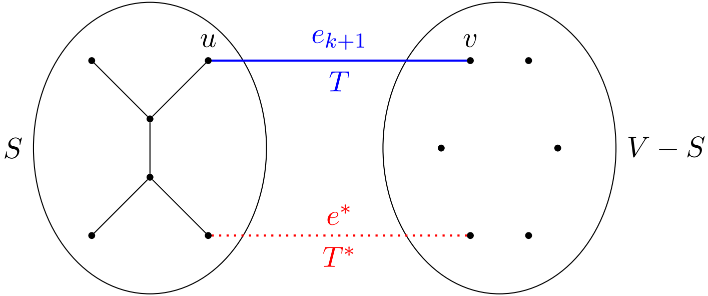


Induction Step (2/2)

Case $e_{k+1} \notin E$: adding e_{k+1} to E would create a cycle between (u, v). In this cycle, $\exists$ another edge e^* connecting S and $V - S$.

Let $T^* = (E - \{e^*\}) \cup \{e_{k+1}\}$, then T^* is also a spanning tree of G, which consists of $e_1, \ldots, e_k, e_{k+1}$.

- If $e_{k+1} < e^*$, then weight$(T^*) <$ weight(T). This contradicts to the hypothesis that T is a MST.
- If $e_{k+1} = e^*$, then weight$(T^*) =$ weight(T). T^* is also a MST, $k+1$ steps outputs is still a subset of T^*.



Kruskal's Algorithm vs. Prim's Algorithm

Kruskal's algorithm

- initial state: $X = \emptyset$, $V(\text{MST}) = \emptyset$
- growth of MST: X always forms a subgraph of final MST
- cut property: try all possible cuts compatible with X
- data structure: union set

Prim's algorithm

- initial state: $X = \emptyset$, $V(\text{MST}) = \forall u$
- growth of MST: X always forms a subtree of final MST
- cut property: select a particular cut determined by X (S is initially an arbitrary vertice, then the vertice set of X)
- data structure: priority queue

Summary of This Lecture

Greedy algorithm. applicable to combinatorial optimization problems: simple and efficient

- build up a solution piece by piece
- alway choose the next piece that offers the most obvious and immediate benefit (rely on heuristic)

How to (dis)prove correctness of greedy algorithm?
(counter-example)

- Mathematical induction (on algorithm steps or input size)
- Exchange argument (gradually transform optimal solution to algorithm solution without affect optimality)

Sometimes greedy algorithm only gives approximate algorithms

Some classical greedy algorithms: Huffman coding, SSSP, MST