DESIGN & ANALYSIS of ALGORITHMS

unit - 4

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Space & Time Tradeoffs

- More likely to trade space for time as there isn't a shortage of space

time efficiency

INPUT ENHANCEMENT

Preprocess problem's input and store additional information obtained to accelerate problem solving speed

(a) Sorting by Counting

I. COMPARISON COUNTING SORTING

- For each element in the list, count number of elements smaller than this element
- The counts indicate the positions of the elements in the sorted array

eg: 62, 31, 84, 96, 19, 47

Maintain a table (array) count

array	62	31	84	96	19	47
count	3	1	4	5	0	2
sorted	19	31	47	62	84	96

Algorithm ComparisonCountingSort (A, n):

```
// Input: unsorted array A
// Output: sorted array S

count[n] = {0}
S[n] = {0}

for i = 0 to n-2:
    for j = i+1 to n-1:

        if A[i] > A[j]:   // a number < A[i] found
            count[i] = count[i] + 1

        else
            count[j] = count[j] + 1

for i = 0 to n-1
    S[count[i]] = A[i]

return S
```

- Time complexity

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} n-1-i \cancel{-1} \cancel{+1} = n-1 + n-2 + \dots + 1 = \frac{n(n-1)}{2}$$

$$\in O(n^2)$$

- Data moved only once even though table construction took time n^2

- For array of 1-bit numbers (0's and 1's), count the number of 0's and place them in the first 'count' locations

eg: 0, 1, 1, 0, 1

sorted: $\underbrace{0, \quad 0}_{\text{count} = 2}$, 1, 1, 1

- Sorts in linear time — O(n)

II. DISTRIBUTION COUNTING SORTING

- Sort a finite set of integers u to l (eg: 0 to 9)

- Count the frequencies of every number (store in a map or array - retrieval time is constant)

- frequency table - distribution of numbers

Symbol	frequency
0	5
1	3
2	1
3	0
4	0
5	2

- if unbounded, 4 bytes x 2^{32} = 16 GB of space needed

- if $u - \ell = 1000$, 4 bytes $\times \sim 2^{10}$ = 4KB of space needed, easily doable
- usable until $\sim 10^6$ entries

eg: $u = 13$, $\ell = 11$

13, 11, 12, 13, 12, 12

symbol	11	12	13
frequency	1	3	2
distribution value	1	4	6

S = 11, 12, 12, 12, 13, 13

Algorithm DistributedCountingSort(A, u, ℓ, n)
// input: unsorted array with elements in finite range
// output: sorted array

```
D[u-ℓ+1] = {0}

for i = 0 to n-1:
    D[A[i]-ℓ] = D[A[i]-ℓ] + 1        // index of element

for i = 1 to u-ℓ:                     // distribution
    D[i] = D[i-1] + D[i]

for i = n-1 to 0:
    j = A[i] - ℓ                      // index in D
    S[D[j]-1] = A[i]                  // index in S = D[j] - 1
    D[j] = D[j] - 1                   // next index in D for same A[i]

return S
```

- Complexity: $O(n)$ if $n > (u-\ell+1)$
- Meaningful only for small, finite ranges

(b) String Matching by Preprocessing

I. BRUTE FORCE

eg: pattern: "TEXT" (length = m), text: below (length = n)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	R	E	A	D		T	E	X	T	B	O	O	K	S
i = 0	T	E	X	T										
i = 1		T	E	X	T									
i = 2			T	E	X	T								
i = 3				T	E	X	T							
i = 4					T	E	X	T						
i = 5						T	E	X	T					

align and slide until match found

- worst case = text length - pattern length + 1 iterations
- 0 to n-m or n-m+1 trials
- Complexity = $O(mn)$

String matching

```
int find(char *text, char *pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int i, j;

    for (i = 0; i <= n - m; ++i) {
        j = 0;
        while ((j < m) && (pattern[j] == text[i+j])) {
            ++j;
        }

        if (j == m) {
            break;
        }
    }

    if (i > n-m) {
        return 0;
    }
    return 1;
}
```

Output

```
→  1-4 String Matching ./find
Enter text:
textbook
Enter pattern:
book
1
→  1-4 String Matching ./find
Enter text:
textbook
Enter pattern:
boy
0
```

II. HORSPOOL'S ALGORITHM

- Start comparing pattern to string from right side
- Check if character in text/string is in pattern or not
- If not present, shift right by length of pattern
- If present, shift such that rightmost occurence of character in pattern aligns with character in text (less than length)

eg 1:

```
          0 1 2 3 4 5 6 7 8
text    : h e i r l o o m s
              ↓
pattern : o o m
          0 1 2
```

i not in pattern, shift by 3

```
          0 1 2 3 4 5 6 7 8
text    : h e i r l o o m s
                    ↓
pattern :       o o m
                0 1 2
```

index of right o

o in pattern, shift by (3-1-1) = 1

```
          0 1 2 3 4 5 6 7 8
text    : h e i r l o o m s
                      ↓
pattern :         o o m
                  0 1 2
```

o in pattern, shift by (3-1-1) = 1

	0	1	2	3	4	5	6	7	8
text :	h	e	i	r	l	o	o	m	s
								↓	
pattern :						o	o	m	
						0	1	2	

m match! ✓

check o

	0	1	2	3	4	5	6	7	8
text :	h	e	i	r	l	o	o	m	s
							↓	↓	
pattern :						o	o	m	
						0	1	2	

o match! ✓

check o

	0	1	2	3	4	5	6	7	8
text :	h	e	i	r	l	o	o	m	s
						↓	↓	↓	
pattern :						o	o	m	
						0	1	2	

o match! ✓

pattern found

Preprocessing: Table of ASCII characters

- Indicates amount to shift by ; shift table
- check if character present in $(m-1)$ characters; last character already compared

eg: pattern = 'algo'

	Shift amount	
a	3	$m-0-1$
b	4	
c	4	
⋮	⋮	
g	1	$m-2-1$
⋮	⋮	
l	2	$m-1-1$
⋮	⋮	
o	4	→ last character

shift amount = $(m-i-1)$

			a
a	l	g	o
0	1	2	3

```
Algorithm ShiftTable(P, m):
  // P: pattern
  // m: length of pattern

  Table[size]        // index in table = value at P[i]

  for i = 0 to size-1:
      Table[i] = m   // length of pattern

  for j = 0 to m-2 :   // last char not compared
      Table[P[j]] = m-j-1

  return Table
```

```
Algorithm Horspool(P, m, T, n):
  // T: text , n: length of text
  // P: pattern , m: length of pattern

  Table = ShiftTable(P, m)      // shift table

  i = m-1      // starting index is rightmost end of pattern

  while i < n:

     k = 0     // number of matched chars from right

     while k<m and P[m-1-k] = T[i-k]
        k=k+1

     if k==m
        return i-m+1

     else
        i= i+ Table[T[i]]

  return -1
```

- Time efficiency
  - Worst case: $\Theta(mn)$
  - Random text: $\Theta(n)$

III. BOYER-MOORE ALGORITHM

- Similar to Horspool
- If first comparison of rightmost character in pattern with corresponding text character fails, shifts right by value from bad symbol table
- Two tables: bad-symbol table, good-suffix table

(a) If first character mismatch, acts as Horspool's and shifts by amount from bad character table

```
. . . m e n l o . . . .
          o n i o n
                    mismatch
```

(b) If k no. of matches occur before the first mismatch is encountered, $(0 < k < m)$

```
. . . l i n t . . .
      v e n t
              k=2
```

text

bad symbol	shift
a	4
:	
e	2
n	1
t	4
v	3

- Shift amount calculated based on two values: bad symbol shift, due to value of first mismatched character from right, and good suffix shift, due to successful match of k characters in the suffix

Bad Symbol Shift

```
. . . h e l i c o p t e r . . .
          s t a r v e r
```

$k=2$

↓ match t

```
. . . h e l i c o p t e r . . .
                s t a r v e r
```

(a) computed as $t_1(c) - k$ where t_1 is bad symbol table, c is first mismatched character in text from the right and k is the number of matched characters

(b) if $t_1(c) - k$ is negative, shift right by 1

```
. . . a c c d e . . .
        b c d e
```

$k=3$ $t_1(c)=2$

- bad symbol shift = $d_1 = \max\{t_1(c) - k, 1\}$

Good Suffix Shift

(a) If suff(k) present in pattern, preceded by a character not equal to c where the mismatch occurred, shift amount by d_2

```
. . . a b c a d . . .
      a d b a d   k = 2
```

suffix

↓ shift by 3

```
. . . a b c a d . . .
            a d b a d
```

(b) If suff(k) does not exist elsewhere in pattern with a different preceding character, find longest prefix of pattern of size $\ell < k$ that matches the suffix of suff(k) and if it exists, shift amount d_2 = distance between them. Otherwise, distance = length of pattern - k

```
. . . a b b a d . . .
      a d b a d   k = 3
```

↓

```
. . . a b b a d . . .
            a d b a d
```

GREEDY TECHNIQUE

- For optimisation problems
- Structure of problems that allow for greedy solutions
- Construction of solution through sequence of steps, each step expanding partial solution to problem
- On each step, choice must be
  - feasible: based on problem constraints
  - locally optimal: best choice locally
  - irrevocable: cannot and need not be undone

Change making Problem

- Provided with an array $D = \{d_1, d_2, \ldots, d_m\}$ of different coin denominations
- Give change of certain amount with least no. of coins
- Subtract largest possible denomination from amount until amount becomes 0
- Eg: $D = \{1, 2, 5, 10\}$ and amount = ₹ 28

$$\left.\begin{array}{l} 28 - 10 = 18 \\ 18 - 10 = 8 \end{array}\right\} 2, 10$$

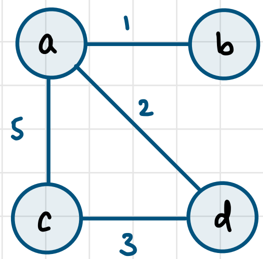
$$8 - 5 = 3 \quad \} \; 1, 5$$

$$3 - 2 = 1 \quad \} \; 1, 2$$

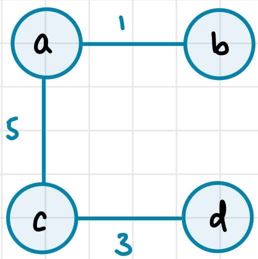
$$1 - 1 = 0 \quad \} \; 1, 1$$

PRIM'S ALGORITHM

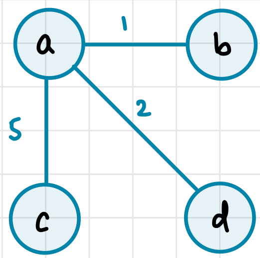
- For general undirected graphs; graph may contain cycles
- Problem: find subgraph of graph (subset of edges) that keeps graph
- Minimum number of edges to connect n nodes = n-1 edges
- Subgraph – spanning tree
- Minimum spanning tree: graph with no cycles of minimum weight
- Eg:



spanning trees:

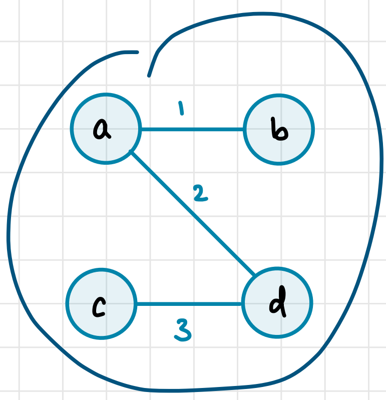


weight = 9



weight = 8

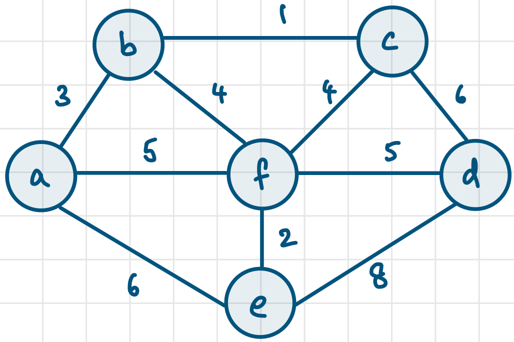
G = MST



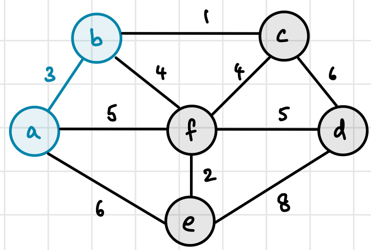
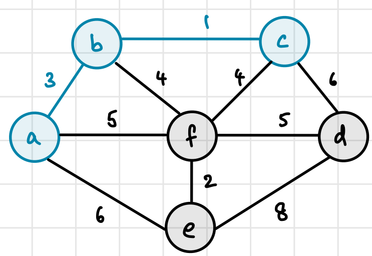
weight = 6

- Update increase in heap — see cormen; $O(\log n)$

Eg: Find an MST of

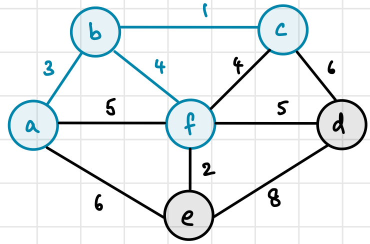


returns list of edges

Tree Vertices	Remaining vertices	Illustration (edges)
a(-, -)	b(a, 3) min c(-, ∞) d(-, ∞) e(a, 6) f(a, 5)	
b(a, 3)	c(b, 1) min d(-, ∞) e(a, 6) f(b, 4)	

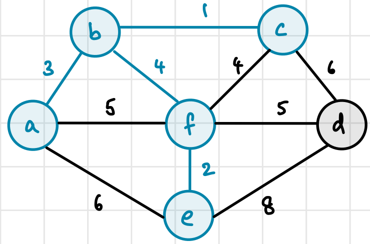
c(b,1)

d(c,6)
e(a,6)
f(b,4) min



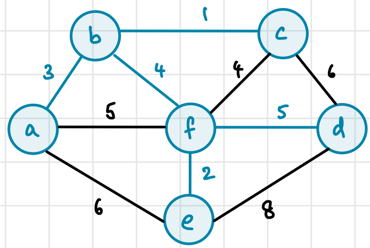
f(b,4)

d(f,5)
e(f,2) min

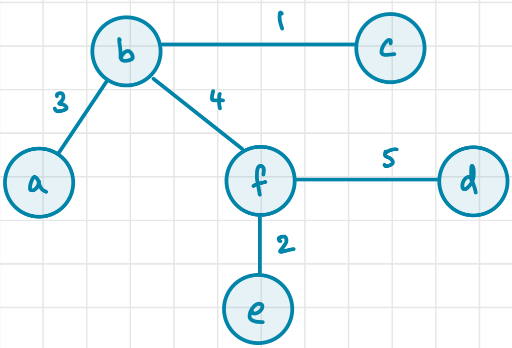


e(f,2)

d(f,5) min



Minimum Spanning Tree



- extract min performed V-1 times for V vertices
- can use heap data structure
- E verifications made
- O(E log V)

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

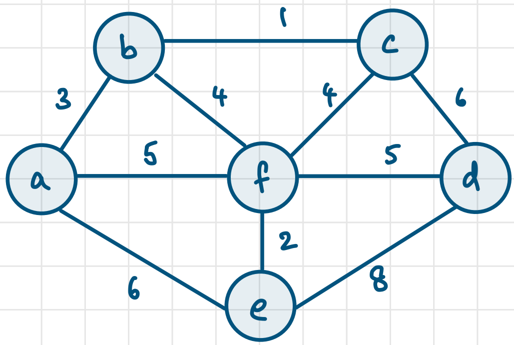
KRUSKAL'S ALGORITHM

- Complementary to Prim's, almost
- Start with forest of isolated nodes (all nodes of graph)
- Gradually merges trees, combining two trees with one edge at every iteration
- All edges are sorted by weight in an increasing order before algorithm runs
- Bottom-up approach (grow nodes to trees)

Algorithm

- $G = \{V, E\}$, $V = \{0, \dots, n-1\}$, $E = \{e_1, \dots, e_n\}$
- Start with empty MST (trees of forest are vertices) and mark each edge as unvisited
- While unvisited edges still present or while MST has less than n-1 edges:
  - Find lightest unvisited edge
  - Mark as visited
  - If adding edge does not create cycle, add to MST

Eg: Find an MST of



returns list of edges

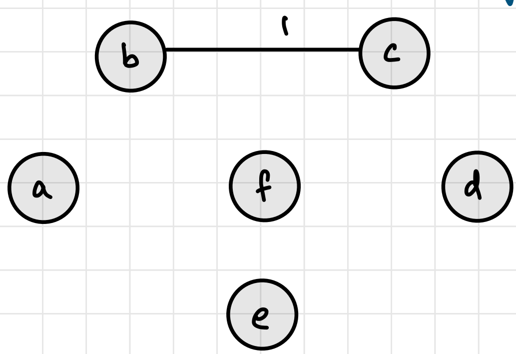
n= 6 , counter = 0

sorted list of edges

Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

1) bc - no cycle

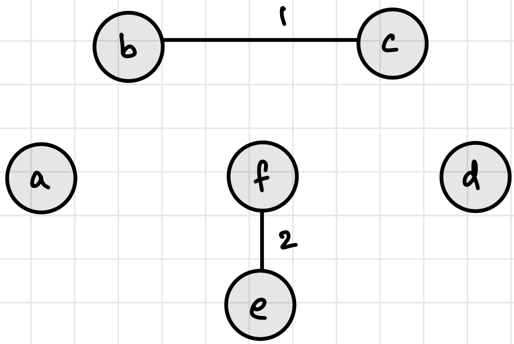
visited & used
visited & discarded



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 1 $\quad$ n-1 = 5

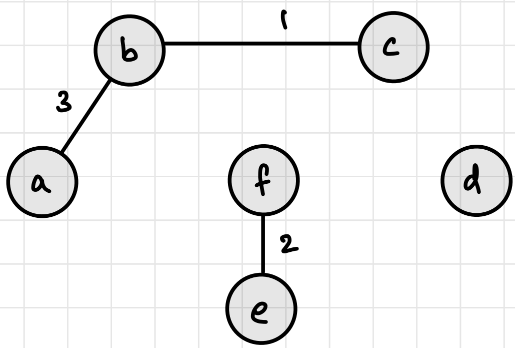
2) ef - no cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 2 $\quad$ n-1 = 5

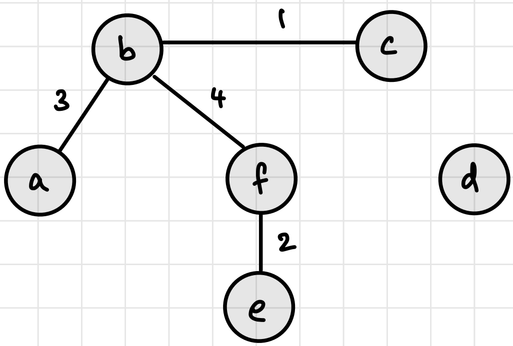
3) ab - no cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 3 $\quad$ n-1 = 5

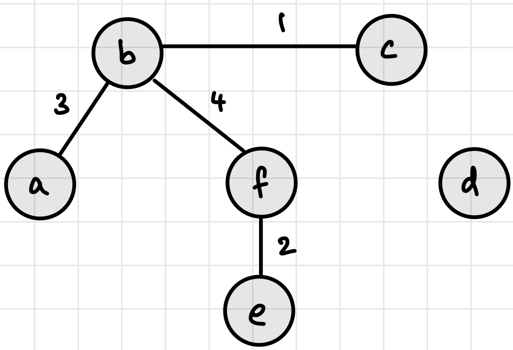
4) bf - no cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 4 n-1 = 5

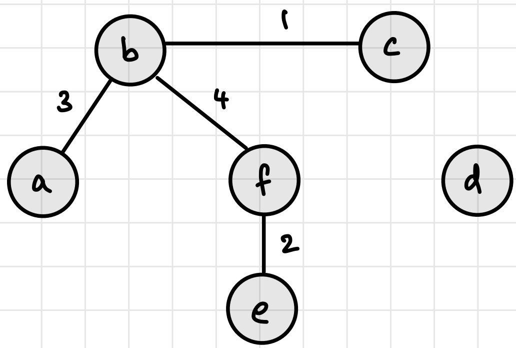
5) cf - cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 4 n-1 = 5

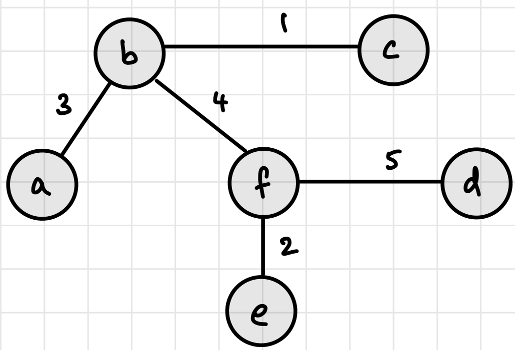
6) af - cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 4   n-1 = 5

7) df - no cycle



Edge (u,v)	bc	ef	ab	bf	cf	af	df	ae	cd	de
Weight	1	2	3	4	4	5	5	6	6	8

counter = 5   n-1 = 5

exit loop

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E\rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  if $E_T \cup \{e_{i_k}\}$ is acyclic
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

DISJOINT SET

- Disjoint set data structure (Union-find)

1. makeset(v): create new set whose only member is pointed to by v. v must already be in a set
2. find (v): returns pointer to set containing v
3. union(u,v): unites dynamic sets that contain u and v into a new set that is union of two sets

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

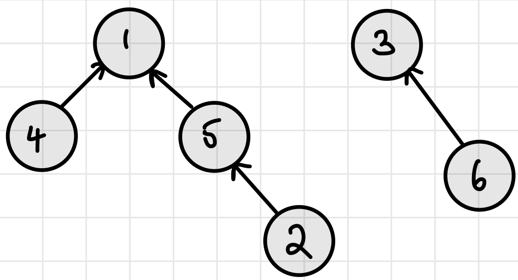
makeset (i) performed 6 times:

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

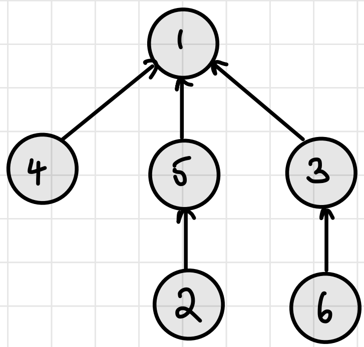
union(1,4) => {1, 4} . . .
union (5,2) => { 5,2} . . .

union (4,5) => {1, 2, 4, 5} . . .
union (3,6) => {3, 6} . . .

forest representation of subsets {1, 2, 4, 5} and {3, 6}



union (5, 6)



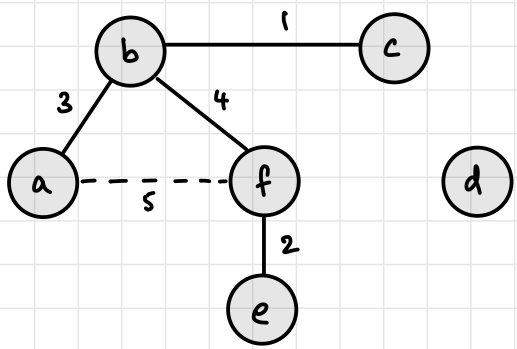
find (4) = find (2) = {1, 2, 3, 4, 5, 6}

To check if $E_T \cup \{e_i\}$ is acyclic,

- perform find (x) and find (y) where xy forms edge e_i
- if find (x) = find (y), cycle will form

eg: $E_T = \{ab, bf, ef, bc\}$

$e_i = af$



find (a) = {a, b, c, e, f}

find (f) = {a, b, c, e, f}

- cycle present, do not include edge

Time complexity

- If union find operation fast enough, determined by sorting algorithm
- $\Theta(|E| \log |E|)$

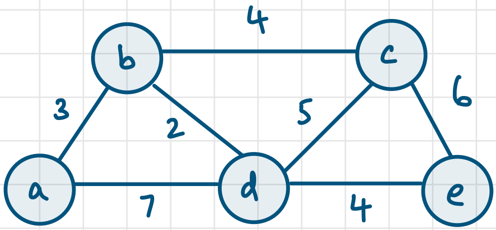
DIJKSTRA'S ALGORITHM

- Single source shortest path problem
- Weighted graph (directed or undirected) G, source vertex s
- Shortest paths from s to all other vertices in the graph
- Similar to Prim's MST algorithm
- Cost of source to source is initialised to 0
- Cost of source to every other vertex initialised to ∞
- Next candidate is the adjacent vertex with lightest edge
- Finds vertex u with smallest sum

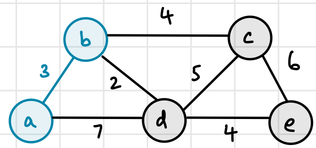
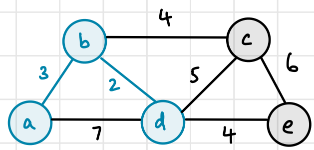
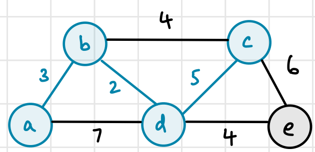
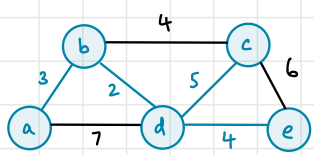
$$d_v + w(v, u)$$

v = vertex for which shortest path already found
d_v = shortest path from s to v
$w(v,u)$ = weight of edge from v to u

Eg:



Source: a

Tree vertices	Remaining vertices	Illustration (edges)
a(−, 0)	b(a, 3) c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3+4) = c(b, 7) d(b, 3+2) = d(b, 5) e(−, ∞)	
d(b, 5)	c(b, 7) e(d, 5+4) = e(d, 9)	
c(b, 7)	e(d, 9)	

Algorithm

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//          and its vertex s
//Output: The length d_v of a shortest path from s to v
//          and its penultimate vertex p_v for every vertex v in V
Initialize(Q)    //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;  p_v ← null        initialise distances to ∞
    Insert(Q, v, d_v)   //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)   //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)   //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```

Time Complexity

- $\Theta(V^2)$ —— adjacency matrix & array priority queue
- $\Theta(E \log V)$ — adjacency list & min heap priority queue

HUFFMAN TREES

- Morse code for e : •
  Morse code for a: • –
  Morse code for q: – – • –
  Morse code for z: – – •

  different lengths

- Most frequently occurring letters: e , a — short
  Least frequently occurring letters: q, z — long

- Huffman encoding: using 0's and 1's — variable length encoding

- ASCII : fixed length encoding

- Prefix encoding : prefixes are unique and assigned

  101 : a
  1010 : b

  not allowed

- Morse code does not use prefix encoding ; gaps after every transmission

Huffman's algorithm

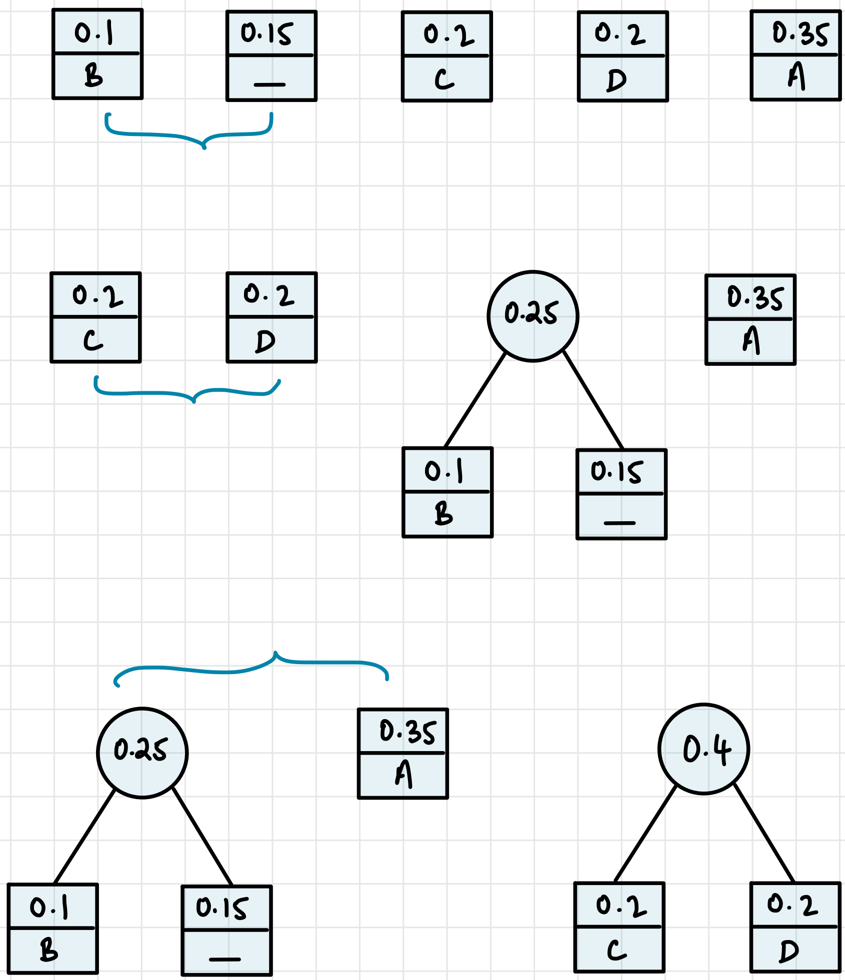
Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

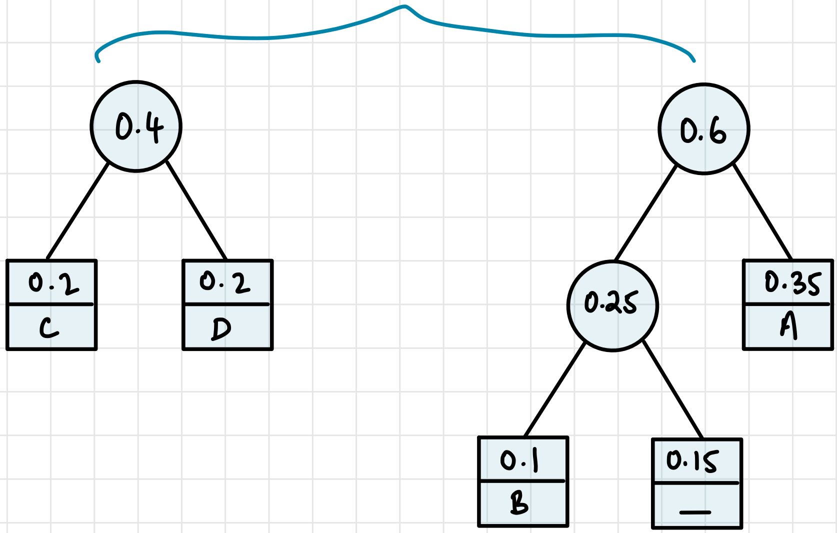
Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

Eg: Consider five symbol alphabet {A, B, C, D, _} with the following occurrence frequencies

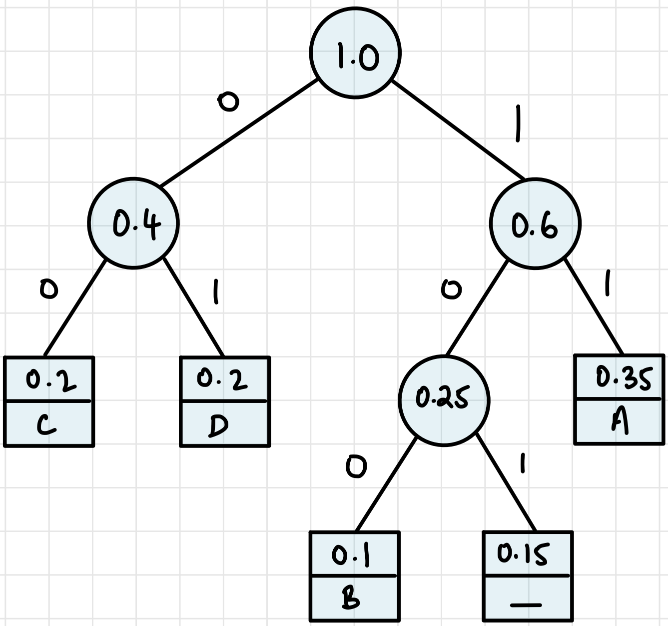
Symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

fixed size encoding: 3 bits





Huffman Tree



Codewords

Symbol	A	B	C	D	_
Codeword	11	100	00	01	101

eg: DAD = 011101

fixed size encoding: 3 bits

avg var length encoding:

$$= 2\times0.35 + 3\times0.1 + 2\times0.2 + 2\times0.2 + 3\times0.15 = 2.25$$

compression ratio $= \frac{3-2.25}{3} = 25\%$

- File compression