Algorithms and Data Structures
Lecture notes: Algorithm paradigms: Greedy algorithms

Lecturer: Michel Toulouse

Hanoi University of Science and Technology
michel.toulouse@soict.hust.edu.vn

29 décembre 2020

Outline

D&C, DP and Greedy algorithms

Divide and conquer algorithm decomposes the problem into subproblems but unfortunately, while doing so, D&C solves some subproblems several times

Bottom up dynamic programming computes the solution of each subproblem, thus solving subproblems that are irrelevant to the solution of the initial problem

Greedy, when it works, only solve subproblems that are relevant to the solution of the whole problem.

Greedy algorithms

With greedy algorithms, one attempts to construct a solution in stages

At each stage a decision is made that appears to be the best at that stage

This decision is made based on a *greedy criterion*

A decision made in one stage is not changed in a later stage, so each decision should assure feasibility.

Greedy algorithms have intuitive appeal, however sometime they fail to compute an optimal solution.

Design features of a greedy algorithm

Before designing a greedy algorithms, we first need to identify 5 features of a problem :

1. A candidate set (objects, in 0-1 knapsack) from which an optimal solution should be built
2. A selection function (greedy criterion) which at each stage selects from the candidate set the most promising element that has not yet been selected
3. A feasibility function checks whether adding a particular candidate satisfies the feasibility constraints, if not discards the candidate and never considers it again
4. A solution function that checks whether the current set of chosen candidates is a complete solution
5. An objective function which assigns value to each partial or complete solution

Example : greedy features for the 0-1 knapsack problem

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

- The 4 objects constitute the candidate set
- 3 well known selection functions (greedy criteria) that apply to the 0-1 knapsack problem :
  1. Income criterion : select the candidate that has the largest value
  2. Weight criterion : select the candidate that has the smallest weight
  3. Profit density criterion : select the largest ratio $\frac{v[i]}{w[i]}$
- Solution function : all objects have been considered
- Feasibility function Considering object j, $\sum_i^4 x_i w[i] + w[j] \leq W$
- Objective function $\max \sum_{i=1}^4 x_i v[i]$

Note, greedy algorithms do not work for the 0-1 knapsack, they provide feasible solutions which are not necessary optimal.

Greedy loop algorithm

- Initially, a set of chosen candidates is empty
- At each step, try to add to this set the best remaining candidate from the candidate set according to a greedy criterion
- If the increased set of chosen candidates is not feasible, remove the candidate just added, it is never considered again. Otherwise, candidate just added stays in the set of chosen candidates
- Each time the set of chosen candidates is increased, check whether the set constitutes a solution to the problem
- When a greedy algorithm works for a problem, the first solution found in this way is always optimal !

A greedy algorithm for 0-1 knapsack

Object i	1	2	3	4	5
Weight w_i	2	3	5	6	4
Value v_i	6	1	12	8	7

Greedy_0-1-Knapsack(int W = 11,int n)
int $selected\_candidates[n]$; int $w = W$; int $candidates\_set[n]$; int i, j
for($i = 1; i \leq n; i++$) $selected\_candidates[i] = 0$
Sort in $candidates\_set[n]$ objects along values, weights or density $\frac{v[i]}{w[i]}$
for $(i = 1; i \leq n, i++)$
  $j = candidates\_set[i]$
  if $(w > w_j)$
    $selected\_candidates[i] = j$
    $w = w - w_j$
  $i++$

Greedy criterion is values

Object i	1	2	3	4	5
Weight w_i	2	3	5	6	4
Value v_i	6	1	12	8	7

Greedy_0-1-Knapsack(int W = 11,int n)
int $selected\_candidates[n]$; int $w = W$; int $candidates\_set[n]$; int i, j
for($i = 1; i \leq n; i++$) $selected\_candidates[i] = 0$
Sort in $candidates\_set[n]$ objects along values
for $(i = 1; i \leq n, i++)$
  $j = candidates\_set[i]$
  if $(w > w_j)$
    $selected\_candidates[i] = j$
    $w = w - w_j$
  $i++$

$candidates\_set = [3, 4, 5, 1, 2]$ after sorting
$i = 1$, $object = 3$, $selected\_candidates = [3, 0, 0, 0, 0]$, $w = 6$
$i = 2$, $object = 4$, $selected\_candidates = [3, 4, 0, 0, 0]$, $w = 0$
Solution is 20

Greedy criterion is weights

Object i	1	2	3	4	5
Weight w_i	2	3	5	6	4
Value v_i	6	1	12	8	7

Greedy_0-1-Knapsack(int W = 11,int n)
int $selected\_candidates[n]$; int $w = W$; int $candidates\_set[n]$; int i, j
for($i = 1; i \leq n; i++$) $selected\_candidates[i] = 0$
Sort in $candidates\_set[n]$ objects along weights
for $(i = 1; i \leq n, i++)$
  $j = candidates\_set[i]$
  if $(w > w_j)$
    $selected\_candidates[i] = j$
    $w = w - w_j$
  $i++$

$candidates\_set = [1, 2, 5, 3, 4]$ after sorting
$i = 1$, $object = 1$, $selected\_candidates = [1, 0, 0, 0, 0]$, $w = 9$
$i = 2$, $object = 2$, $selected\_candidates = [1, 2, 0, 0, 0]$, $w = 6$
$i = 3$, $object = 5$, $selected\_candidates = [1, 2, 5, 0, 0]$, $w = 2$
Solution is 14

Greedy criterion is density

Object i	1	2	3	4	5
Weight w_i	2	3	5	6	4
Value v_i	6	1	12	8	7

Greedy_0-1-Knapsack(int W = 11,int n)
int $selected\_candidates[n]$; int $w = W$; int $candidates\_set[n]$; int i, j
for($i = 1; i \leq n; i++$) $selected\_candidates[i] = 0$
Sort in $candidates\_set[n]$ objects along density $\frac{v[i]}{w[i]}$
for $(i = 1; i \leq n, i++)$
  $j = candidates\_set[i]$
  if $(w > w_j)$
    $selected\_candidates[i] = j$
    $w = w - w_j$
  $i++$

$candidates\_set = [1, 3, 5, 4, 2]$ after sorting
$i = 1$, $object = 1$, $selected\_candidates = [1, 0, 0, 0, 0]$, $w = 9$
$i = 2$, $object = 3$, $selected\_candidates = [1, 3, 0, 0, 0]$, $w = 4$
$i = 3$, $object = 5$, $selected\_candidates = [1, 2, 5, 0, 0]$, $w = 0$
Solution is 25

Greedy algo for the making change problem

Features of a greedy algo for the making change problem :

1. Candidate set : a finite set of different types of coins, containing enough coins of each type
2. Greedy criterion : chose the highest-valued coin remaining in the set of candidates
3. Solution function : the total value of the chosen set of coins is exactly the amount we have to pay back
4. Feasibility function : the total value of the chosen set *does not exceed* the amount to be returned
5. Objective function : minimize the number of coins used in the solution.

Making change : a greedy algorithm

Design an algorithm to return back change to an customer for a certain amount n using the smallest possible number of coins of different denominations.

There is an unlimited set of coins of each denomination

Here is a greedy recursive algorithm.

Greedy_MakingChange(int n)
while ($n > 0$)
  $c =$ the largest coin that is no larger than n
  return $\lfloor n/c \rfloor$ of the c cent coins
  Greedy_MakingChange($n - \lfloor n/c \rfloor c$)

Example of making change

Assume the coin denominations are 25, 10, 5 and 1 cents and we make change for 97 cents using the above greedy algorithm.

- Choose the largest coin that is no larger than n (here 25 cents is the largest coin no larger than 97 cents)
- Give out $\lfloor n/c \rfloor$ of the c cent coins ($\lfloor 97/25 \rfloor =$ three quarters)
- Make change recursively for $n - \lfloor n/c \rfloor c$ cents ($97 - 3 \times 25 = 22$ cents)
  - 10 cents is the largest coin no larger than 22 cents, so we hand out two dimes and make change for $22 - 20 = 2$ cents
    - a penny is the largest coin no larger than 2 cents, so we hand out two pennies and we're done!

Altogether we hand out 3 quarters, 2 dimes and 2 pennies, for a total of 7 coins

Greedy may fail to solve making change optimally

The greedy algorithm always gives out the fewest number of coins in change, when the coins are quarters, dimes, nickels and pennies.

It doesn't give out the fewest number of coins if the coins are 15 cents, 7 cents, 2 cents and 1 cent.

Example : For 21 cents the greedy algorithm would

- give out one 15-cent coin, remainder $21 - 15 = 6$ and
- give out three 2-cent coins,
- for a total of four coins.

Better : give out three 7-cent coins for a total of three coins !

Problems where greedy algorithms fail

We have just found two problems, 0-1 knapsack and the making change problem, where greedy algorithms could fail to find an optimal solution

Both problems exhibit the optimal substructure, the optimal solution to a problem instance contains in it the optimal solution of subproblems

While optimal substructure is a necessary condition a problem must satisfy for greedy algorithms to apply, this condition is not sufficient

A second condition needs to be satisfied : the greedy choice property

Greedy choice property

Sometime there are too many subsolutions to check

The "greedy choice property" means that an optimal solution can be obtained by making a "greedy" decision at every step

- In other words there exists a sequence a greedy choices leading to the optimal solution

Don't need to know the solutions of all the subproblems in order to make a choice, always make the choice that looks best at each stage

When we can prove that a problem can be solved by making greedy choices, then greedy is an extremely efficient algorithm !

Minimum Spanning Tree (MST)

Let $G = (V, E)$ be a connected, weighted graph.

A weighted graph is a graph where a real number called weight is associated with each edge.

A spanning tree of G is a subgraph T of G which is a tree that spans all vertices of G. In other words, T contains all of the vertices of G.

Minimum Spanning Tree

The weight of a spanning tree T is the sum of the weights of its edges. That is,

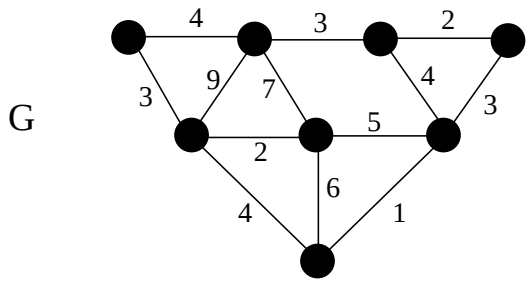
$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

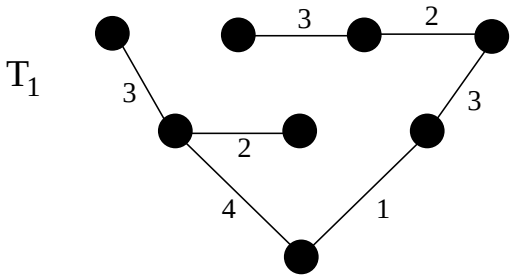
A minimum spanning tree (MST) of G is spanning tree T of minimum weight.

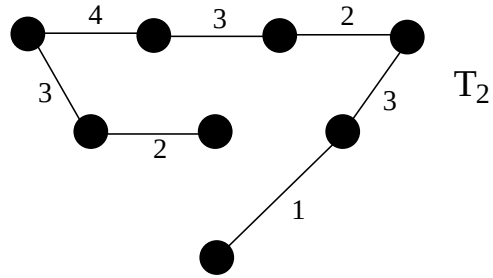
It should be clear that a minimum spanning tree always exists.

Minimum Spanning Tree : Examples

- Here is an example of a graph and two minimum spanning trees.







Optimal substructure for MST

MST exhibits the optimal substructure property : an optimal tree is composed of optimal subtrees

- Let T be an MST of G and consider an edge (u, v) not adjacent to a leave of T
- Removing (u, v) partitions T into two trees T_1 and T_2

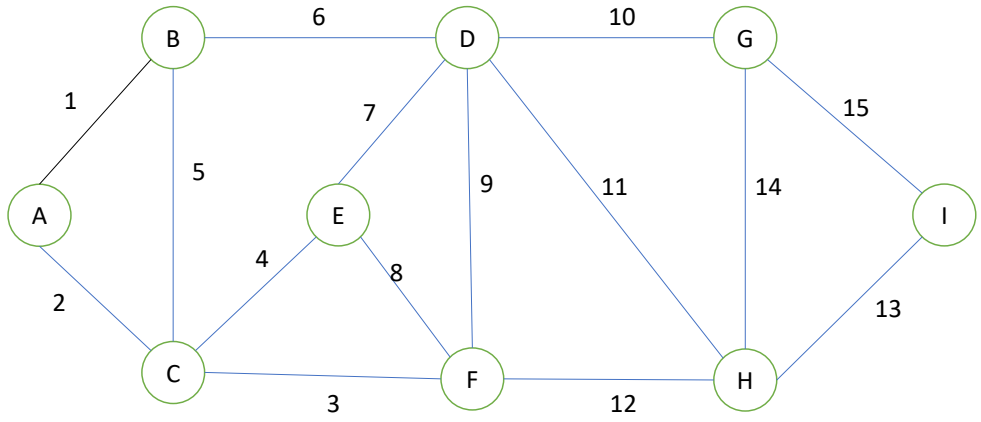
Claim :

T_1 is an MST of $G_1 = (V_1, E_1)$, and T_2 is an MST of $G_2 = (V_2, E_2)$

Proof : Note that $V_1 \cap V_2 = \emptyset$. By construction $w(T) = w(u, v) + w(T_1) + w(T_2)$. If T_1 or T_2 was not a MST then $w(T)$ would be suboptimal, a contradiction of the assumption that T is a MST.

Greedy choice property for MST

Let $G = (V, E)$ be an undirected graph. Let e_v be the cheapest edge adjacent to each vertex v.



Theorem :

The minimum spanning tree of G contains every e_v

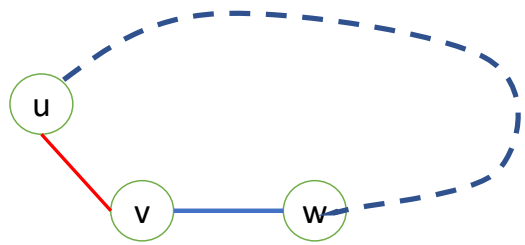
Greedy choice property for MST

Proof :

Suppose on the contrary that MST of G does not contain some edge $e_v = (u, v)$.

Let T = optimal MST of G

By adding $e_v = (u, v)$ to T, we obtain a cycle

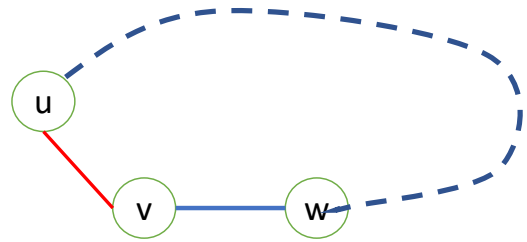


Greedy choice property for MST

Proof :

By our choice of e_v, the weight of (u, v) must be less than the weight of (v, w) in T

If we delete (v, w) and include e_v, we obtain a new spanning tree cheaper than T, a contradiction



Constructing an MST

The MST problem exhibit the optimal substructure and greedy properties.

Therefore minimum spanning tree problem can be solved optimally using a greedy algorithm.

There are actually two common greedy algorithms to construct MSTs :

- **Kruskal's algorithm**
- **Prim's algorithm**

Both of these algorithms use the same basic ideas, but in a slightly different fashion.

Kruskal's algorithm background

Kruskal's algorithm is a greedy algorithm to compute minimum spanning trees of a n nodes graph G

This algorithm first creates n different trees each containing only one node from G

It then sorts the edges of G in increasing order of their length and stores them in an array S

A greedy **for** loop then selects edges from S, each edge connects a pair of nodes, thus generating a new tree built from the trees corresponding to the two nodes.

This process continues until there is only one tree containing all the nodes from G

Greedy features of Kruskal's algorithm

Candidate set : the edges from G

Selection function (greedy criterion) : $\{u, v\}$, shortest edge not yet considered

Solution function Does the selected edges form a spanning tree of G

Feasibility function Does adding the current edge create a cycle

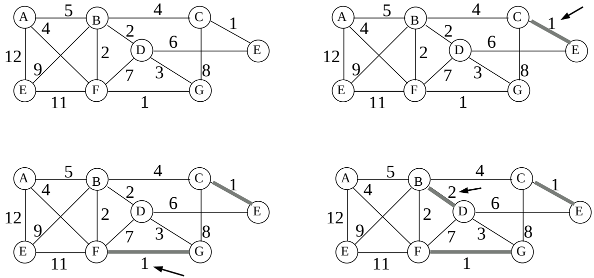
Objective function : Minimize the sum of the weights of the edges in T

Kruskal's Algorithm

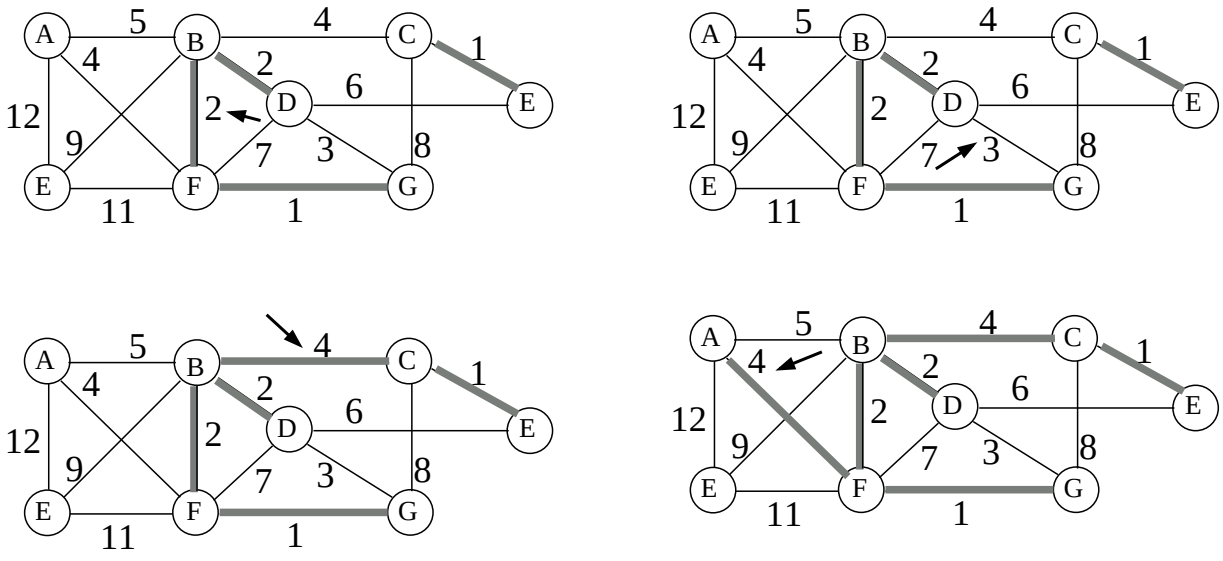
Algorithm Kruskal(G)
input : Graph $G = (V, E)$;
output : A minimum spanning tree T;
  Sort E by increasing *length*;
  $n = |V|$;
  $T = \emptyset$;
  For each $v \in V$ MakeSet(v)
  for $i = 1$ to $|E|$ **do** /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered;
    if (FindSet(u) $\neq$ FindSet(v)) then
      Union(FindSet(u), FindSet(v))
      $T = T \bigcup \{u, v\}$;
  return T

Kruskal's Algorithm Example

for $i = 1$ to $|E|$ **do** /* Greedy loop */
  $\{u, v\}$ = shortest edge not yet considered ;
  if (FindSet(u) $\neq$ FindSet(v)) then
    Union(FindSet(u), FindSet(v))
    $T = T \bigcup \{u, v\}$;
return T



for $i = 1$ to $|E|$ **do** /* Greedy loop */
  $\{u, v\}$ = shortest edge not yet considered ;
  if (FindSet(u) $\neq$ FindSet(v)) then
    Union(FindSet(u), FindSet(v))
    $T = T \bigcup \{u, v\}$;
return T



for $i = 1$ to $|E|$ **do** /* Greedy loop */

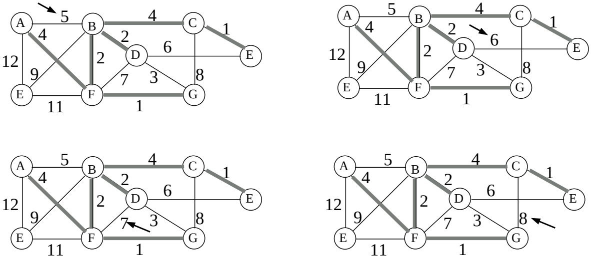
  $\{u, v\}$ = shortest edge not yet considered ;

  if (FindSet(u) $\neq$ FindSet(v)) then

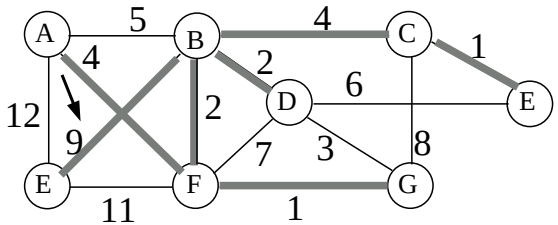
    Union(FindSet(u), FindSet(v))

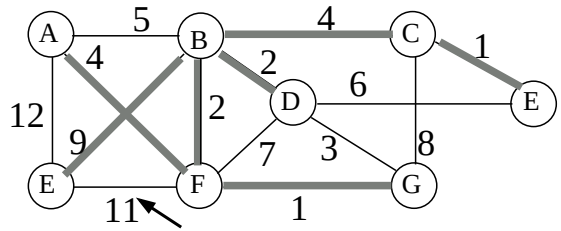
    $T = T \bigcup \{u, v\}$;

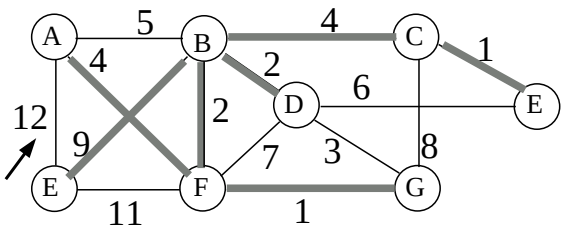
return T

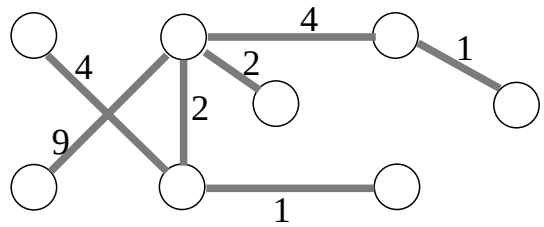


for $i = 1$ to $|E|$ **do** /* Greedy loop */
  $\{u, v\}$ = shortest edge not yet considered ;
  if (FindSet(u) $\neq$ FindSet(v)) then
    Union(FindSet(u), FindSet(v))
    $T = T \bigcup \{u, v\}$;
return T









Kruskal's Algorithm : time complexity analysis

Algorithm Kruskal(G)
input : Graph $G = (V, E)$;
output : A minimum spanning tree T;
  Sort E by increasing *length*;
  $n = |V|$;
  $T = \emptyset$;
  for each $v \in V$ MakeSet(v)
  for $i = 1$ to $|E|$ **do**
    shortest $\{u, v\}$ not yet considered;
    if (FindSet(u) $\neq$ FindSet(v)) then
      Union(FindSet(u), FindSet(v))
      $T = T \bigcup \{u, v\}$;
  return T

- Sort edges : $O(E \lg E)$
- $O(n)$ MakeSet()'s
- $O(E)$ FindSet()'s Union()'s operations

Exercises on Kruskal's algorithm

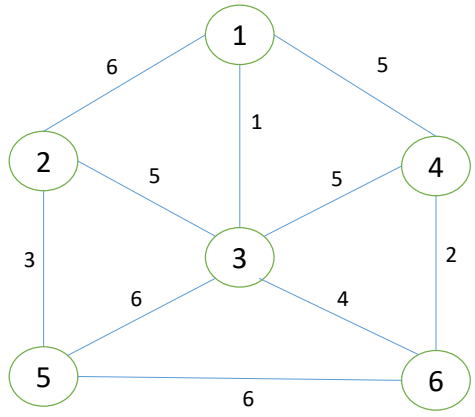
Consider the non-oriented weighted graph G below represented using an adjacency matrix

	a	b	c	d	e	f	g	h	i	j	k	l	m
a		1		6	2								
b			1			2	3						
c							4						
d							1	5	3	1			
e						1	3	2					
f							2						
g								4					
h										3	1		
i										2			
j											1		
k												1	
l													2
m													

1. Draw this graph
2. Compute by hand a minimum spanning tree using Kruskal's algorithm. Show which edge is selected at each step of the greedy algorithm and tell whether it is included in the spanning tree or whether it is rejected
3. Is there more than one minimum spanning for this graph

Exercise on Kruskal's algorithm

Compute the MST using Kruskal's algorithm. At each step of the computation, show the configuration of sub-trees.



Prim's Algorithm Background

Unlike Kruskal's algorithm, Prim's algorithm grows a single tree T into a minimum spanning tree

An arbitrary vertex r is picked, and the tree is grown from that vertex

Thus, we add a vertex and an edge to T at each step

Since T is a tree, it remains a tree with the added edge and node

Greedy aspects of Prim's algorithm

The greedy aspects of Prim's algorithm are the same as for Kruskal's algorithm except for the greedy criterion

Greedy criterion : Select the edge that has minimum cost that has one of its endpoint in the current spanning tree.

Prim's Algorithm data structures

For each node x, we store

- The predecessor $p(x)$. This is the vertex y in T which we join x to when edge (x, y) is added to T.
- The $key(x)$. The weight of the minimum weight edge that connects x to some vertex in T.

$key(r) = 0$, and $p(r) = NIL$ throughout.

Each node $x \neq r$ starts with $key(x) = \infty$.

The value $key(x)$ only changes if some node adjacent to x is added to T.

Prim's Algorithm–More Details

Thus, when a node y is added to T, the key values of the nodes adjacent to y is updated

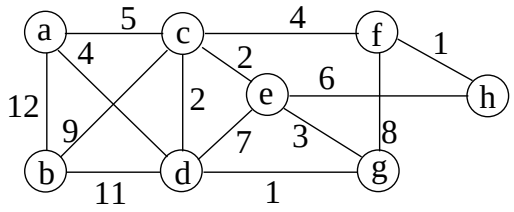
The vertices in $V - T$ are stored in a priority queue Q based on $key(x)$. This allows to pick the minimum weight edge to add to T.

T is not stored explicitly. The MST is reconstructed using the predecessors $p(x)$ for all $p \neq r$.

Prim's Algorithm

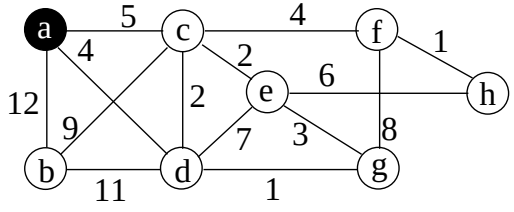
```
Prim_MST(G, r)
  ∀ u ∈ G
    key[u] = Max_Int;
  key[r] = 0;
  p[r] = NIL;
  Q = MinPriorityQueue(V)
  while (Q ≠ ∅)
    u = ExtractMin(Q);
    for each v adjacent to u
      if ((v ∈ Q) & (w(u, v) < key[v]))
        p[v] = u;
        key[v] = w(u, v);
```

```
while (Q ≠ ∅)
  u = ExtractMin(Q);
  for each v adjacent to u
    if ((v ∈ Q)
    & (w(u, v) < key[v]))
      p[v] = u;
      key[v] = w(u, v);
```



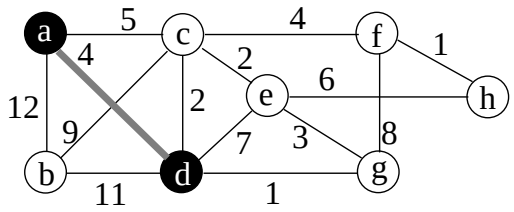
v	a	b	c	d	e	f	g	h
k	0	∞	∞	∞	∞	∞	∞	∞
p	nil	?	?	?	?	?	?	?

Q=[a,b,c,d,e,f,g,h]



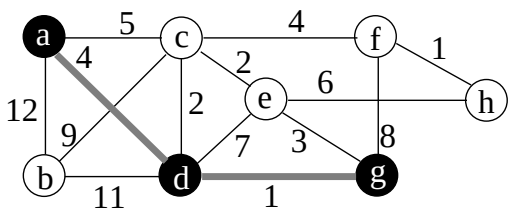
v	a	b	c	d	e	f	g	h
k	~~0~~	12	5	4	∞	∞	∞	∞
p	nil	a	a	a	?	?	?	?

Q=[d,c,b,e,f,g,h]



v	a	b	c	d	e	f	g	h
k	~~0~~	11	2	~~4~~	7	∞	1	∞
p	nil	d	d	a	d	?	d	?

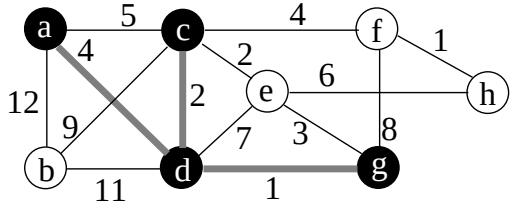
Q=[g,c,e,b,f,h]



v	a	b	c	d	e	f	g	h
k	~~0~~	11	2	~~4~~	3	8	~~1~~	∞
p	nil	d	d	a	g	g	d	?

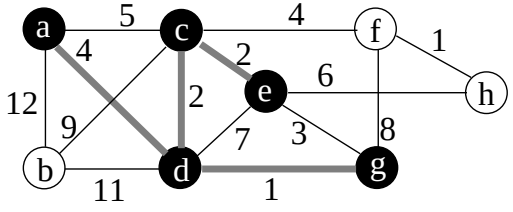
Q=[c,e,f,b,h]

```
while (Q ≠ ∅)
   u = ExtractMin(Q);
   for each v adjacent to u
      if ((v ∈ Q)
      & (w(u, v) < key[v]))
         p[v] = u;
         key[v] = w(u, v);
```



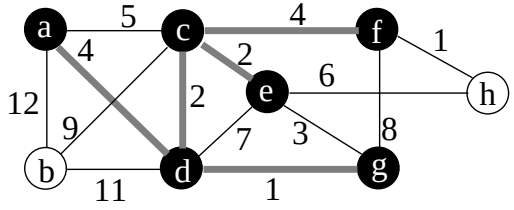
v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	2	4	~~1~~	∞
p	nil	c	d	a	c	c	d	?

Q=[e,f,b,h]



v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	~~2~~	4	~~1~~	6
p	nil	c	d	a	c	c	d	e

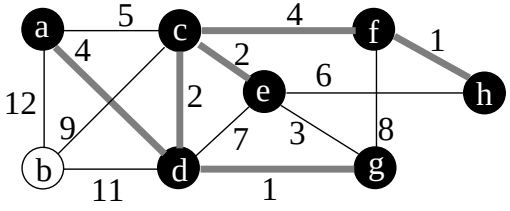
Q=[f,h,b]



v	a	b	c	d	e	f	g	h
k	~~0~~	9	~~2~~	~~4~~	~~2~~	~~4~~	~~1~~	1
p	nil	c	d	a	c	c	d	f

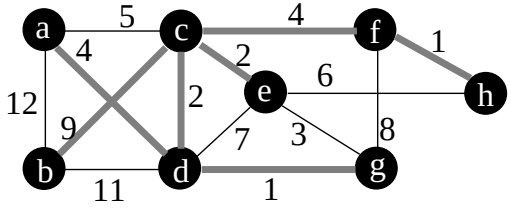
Q=[h,b]

```
while (Q ≠ ∅)
    u = ExtractMin(Q);
    for each v adjacent to u
        if ((v ∈ Q)
        & (w(u, v) < key[v]))
            p[v] = u;
            key[v] = w(u, v);
```



v	a	b	c	d	e	f	g	h
k	~~∞~~	9	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
p	nil	c	d	a	c	c	d	f

Q=[b]



v	a	b	c	d	e	f	g	h
k	~~∞~~	~~9~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
p	nil	c	d	a	c	c	d	f

Q=[]

Prim's Algorithm : time complexity analysis

```
Prim_MST(G, r)
  ∀ u ∈ G
    key[u] = Max_Int;
  key[r] = 0;
  p[r] = NIL;
  Q = MinPriorityQueue(V)
  while (Q ≠ ∅)
    u = ExtractMin(Q);
    for each v adjacent to u
      if ((v ∈ Q) & (w(u, v) < key[v]))
        p[v] = u;
        key[v] = w(u, v);
```

Priority Queue : $O(n)$

while loop runs n times

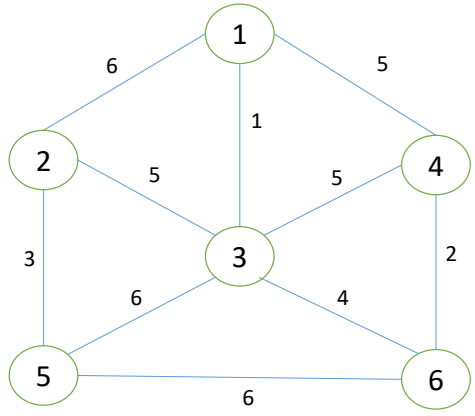
- ExractMin cost $O(\lg n)$, total $O(n \lg n)$
- **for** loop is executed $2|E|$ times (Amortized analysis)
- Each time could potentially change $key[v]$, $O(\lg n)$
- Total cost of **for** loop is $O(E \lg n)$

while loop is

$O(n \lg n + E \lg n) \in O(E \lg n)$

Exercise on Prim's algorithm

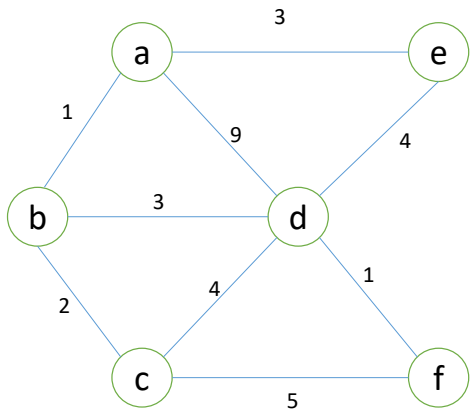
Compute the MST using Prim's algorithm. At each step of the computation, show the configuration of the partial MST.



Exercise on Prim's algorithm

Compute the MST using Prim's algorithm. Fill the table at each step.

v	a	b	c	d	e	f
k						
P						

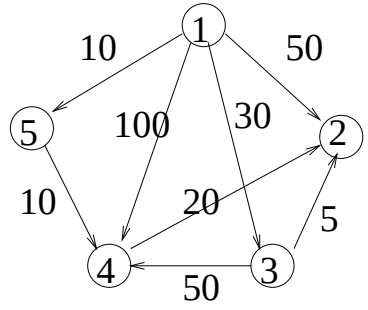


The single-source shortest paths problem

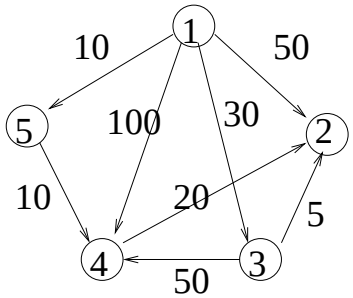
Let $G = \{V, E\}$ be a connected directed graph where V is the set of nodes and E is the set of arcs.

- Each arc $a \in E$ has a nonnegative length.
- One of the node is designated as the source node.
- The problem is to determine the length of the shortest path from the source node to each of the other nodes of the graph

This problem can be solved by a greedy algorithm called Dijkstra's algorithm



Dijkstra's algorithm



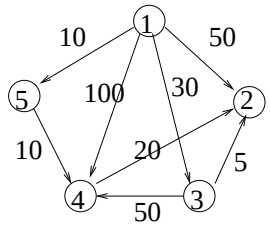
Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

C is the candidate set which initially contains the nodes of G minus the source node

S be the set of selected nodes, i.e. the nodes whose minimal distance from the source is already known

The length of each arc is stored in an $n \times n$ distance matrix L

Shortest special path



Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

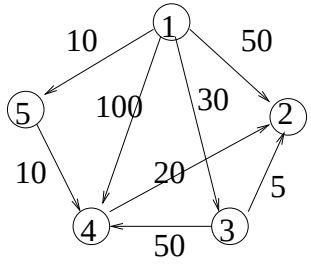
A path from the source to another node is special if all intermediate nodes along the path belong to S

Let D be an array such that at each step of the algo., D contains the length of the shortest special path to each node in the graph.

Initially, only the source node is in S, so the entries of D are the length of the direct edge from the source to each node

When a new node v is added to S, the shortest special path to v is also the shortest of all the paths to v

Dijkstra's algorithm



Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

Algorithm Dijkstra(G)
  $C = \{2, 3, \ldots, n\}$ $\{S = V \setminus C\}$
  for $i = 2$ **to** n **do** $D[i] = L[1, i]$
  repeat $n - 2$ **times**
    $v =$ some element of C minimizing $D[v]$
    $C = C \setminus \{v\}$
    for each $w \in C$ **do**
      $D[w] = min(D[w], D[v] + L[v, w])$
  return D

Dijkstra's algorithm : time complexity analysis

Algorithm Dijkstra(G)
  $C = \{2, 3, \ldots, n\}$ $\{S = V \setminus C\}$
  for $i = 2$ **to** n **do** $D[i] = L[1, i]$
  repeat $n - 1$ **times**
    $v =$ some element of C minimizing $D[v]$
    $C = C \setminus \{v\}$
    for each $w \in C$ **do**
      $D[w] = min(D[w], D[v] + L[v, w])$
  return D

The initialization of C and D cost $O(n)$

In the **repeat** loop

- the instruction "$v =$ some element of C minimizing $D[v]$" cost $O(n)$
- The **for** loop is also running in $O(n)$

Therefore the **repeat** loop runs in $O(n^2)$.

Dijkstra's algorithm : time complexity analysis

If $G = (V, A)$ is not dense, i.e. $|E|$ much smaller than $\frac{n \times n-1}{2}$, then we can use a min heap containing one node for each $v \in C$ ordered by the value of $D[v]$ (thus $v \in C$ that minimizes $D[v]$ will always be at the root)

Initializing the heap take $O(n)$ (BuildHeap)

The instruction "$v =$ some element of C minimizing $D[v]$" cost $O(\log n)$ (Heapify $O(logn)$)

In the **for** loop, if $D[w]$ change, we must move w up in the heap, $O(logn)$. Using amortized analysis, we can show that Heapify will not be called more than once for each edge, therefore $O(|E|log\ n)$.

The total cost of the **repeat** loop is $O(|E| + n \log n)$, since the graph is connected, we have $|E| \geq n - 1$, therefore **repeat** loop is $O(|E| \log n)$

Dijkstra's algorithm : obtaining the shortest paths

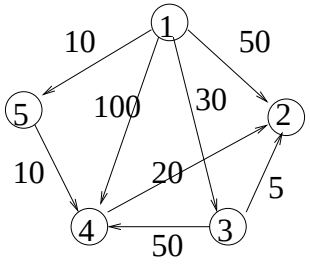
To obtain the shortest paths (not just their length), we need a new array $P[2..n]$ where $P[v]$ contains the node that precedes v in the shortest path

To find the shortest path from a node v to the source just follow the pointers in reverse direction starting at v

Algorithm Dijkstra(G)
  $C = \{2, 3, \ldots, n\}$ $\{S = V \setminus C\}$
  for $i = 2$ **to** n **do**
    $D[i] = L[1, i]$
    $P[i] = 1$
  repeat $n - 1$ **times**
    $v =$ some element of C minimizing $D[v]$
    $C = C \setminus \{v\}$
    for each $w \in C$ **do**
      if $D[w] > D[v] + L[v, w]$ **then**
        $D[w] = min(D[w], D[v] + L[v, w])$
        $P[w] = v$
  return D and P

Dijkstra's algorithm : obtaining the shortest paths

Here are consecutive states of P for the previous example :



Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

- $v = 5$

P	1	2	3	4	5
	1	1	1	5	1

- $v = 4$

P	1	2	3	4	5
	1	4	1	5	1

- $v = 3$

P	1	2	3	4	5
	1	3	1	5	1

- $v = 2$

P	1	2	3	4	5
	1	3	1	5	1

Optimal substructure property

First we prove that single-source shortest paths problem satisfies the optimal substructure property, i.e. a subpath of a shortest past is a shortest path as well.

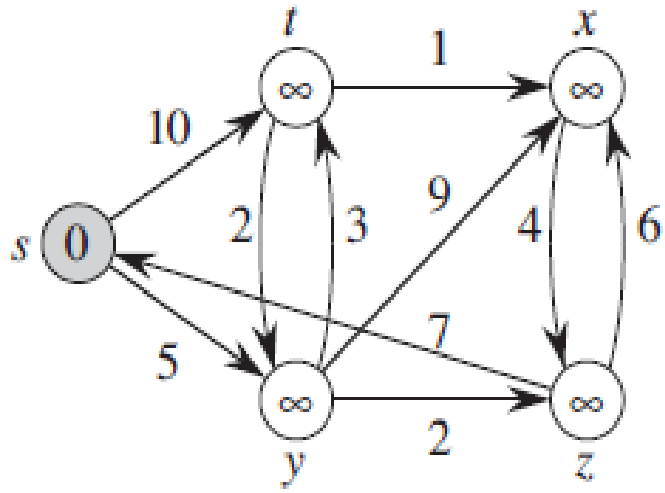
Let $G = (V, E)$ be a weight directed graph with only positive weights. Let $p = [0, 1, \ldots, k]$ be the shortest path between nodes 0 and k in G.

Theorem 2 : Let p_{ij} be a subpath from node i to node j of the path p. Then p_{ij} is the shortest path between nodes i and j in G.

Proof : If we decompose p into subpaths p_{0i}, p_{ij} and p_{jk} then the cost $w(p)$ of the path p is $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Assume there is a path p'_{ij} between nodes i and j such that $w(p'_{ij}) < w(p_{ij})$. Then $(w(p_{0i}) + w(p'_{ij}) + w(p_{jk})) < w(p)$ which contradicts that p is the shortest path between nodes 0 and k.

Exercise : single-source shortest paths

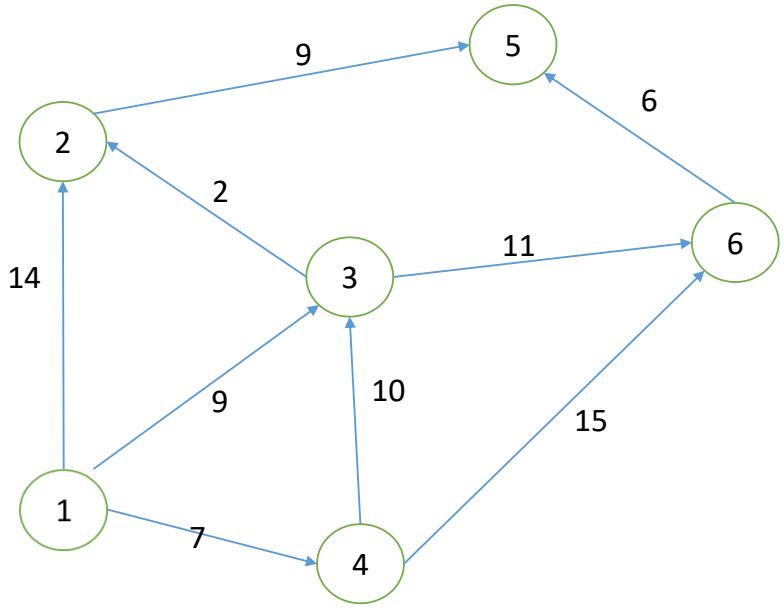
Compute the single-source shortest paths for the oriented graph below. The source is node s.



Algorithm Dijkstra(G)
  $C = \{2, 3, \ldots, n\}$ $\{S = V \setminus C\}$
  for $i = 2$ **to** n **do** $D[i] = L[1, i]$
  repeat $n - 1$ **times**
    $v =$ some element of C minimizing $D[v]$
    $C = C \setminus \{v\}$
    for each $w \in C$ **do**
      $D[w] = min(D[w], D[v] + L[v, w])$
  return D

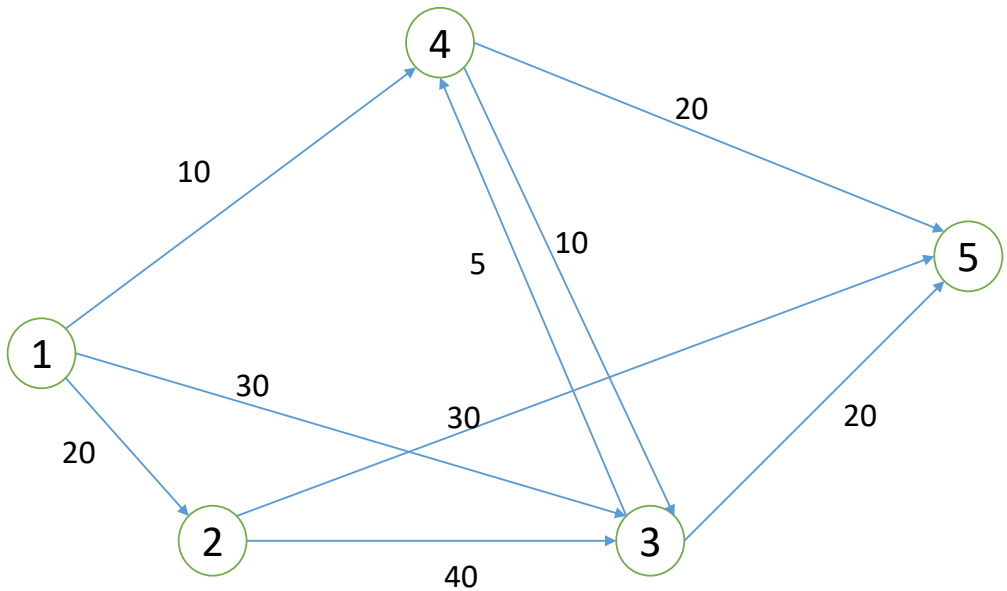
Exercise : single-source shortest paths

Compute the length as well as the single-source shortest paths for the oriented graph below. The source is node 1.



Exercise : single-source shortest paths

Compute the length as well as the single-source shortest paths for the oriented graph below. The source is node 1.



Prim-Dijkstra algorithm

```
Prim_Dijkstra(G, r)
  ∀ u ∈ G
    key[u] = Max_Int ;
  key[r] = 0 ;
  p[r] = NIL ;
  Q = MinPriorityQueue(V)
  while (Q ≠ ∅)
    u = ExtractMin(Q) ;
    for each v adjacent to u
      if ((v ∈ Q) & (f(u, v) < key[v]))
        p[v] = u ;
        key[v] = f(u, v) ;
```

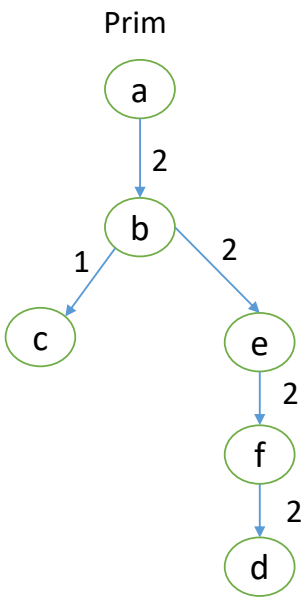
The algorithm Prim_Dijkstra is a generic code that can compute either Prim or Dijkstra. Which is computed depends on the implementation of the function $f(u, v)$:

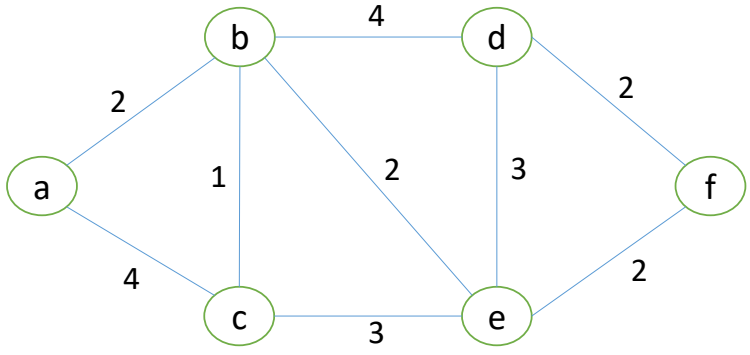
- If Prim then f returns $w(u, v)$
- If Dijkstra then f returns $key[u] + w(u, v)$

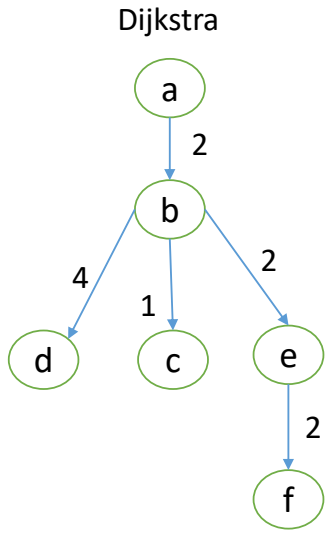
Prim-Dijkstra algorithm

Prim's algorithm computes a tree, so does Dijkstra algorithm

While the tree generated by Prim is an MST, the tree generated by Dijkstra represents the shortest paths in the graph. See this example





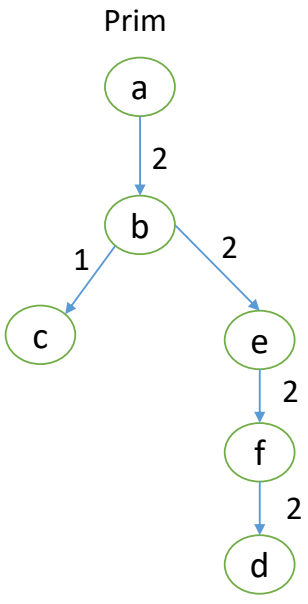


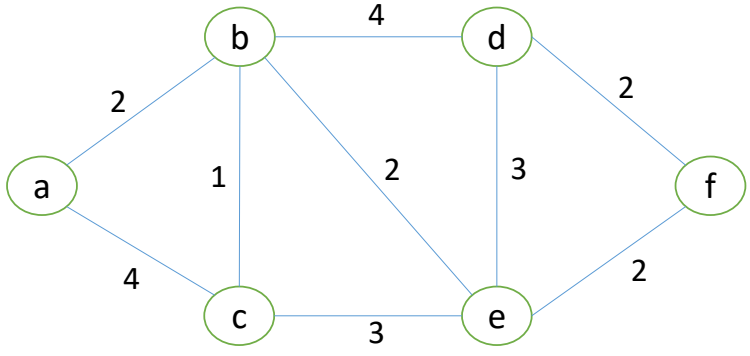
Prim-Dijkstra algorithm

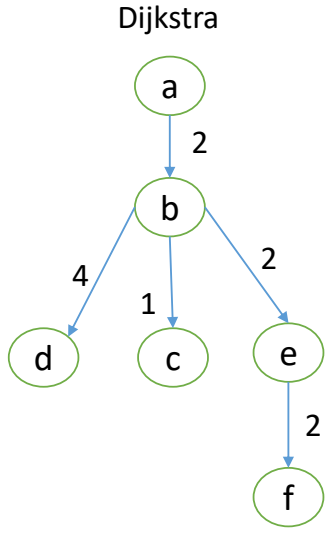
For Prim $W(T) = 2 + 1 + 2 + 2 + 2 = 9$.

For Dijkstra, $W(T) = 2 + 4 + 1 + 2 + 2 = 11$

On the other hand, the path to d in Dijkstra is the shortest one $= 6$, while in tree generated by Prim, the length of the path from a to d is 8.

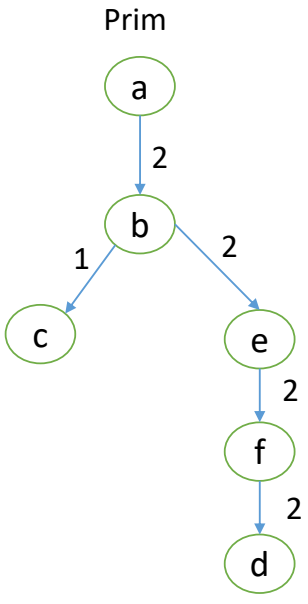


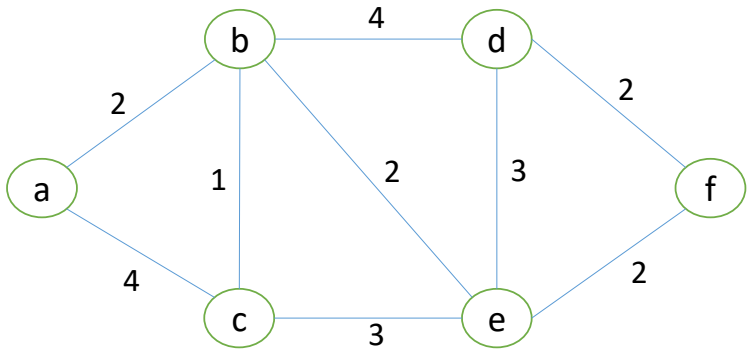


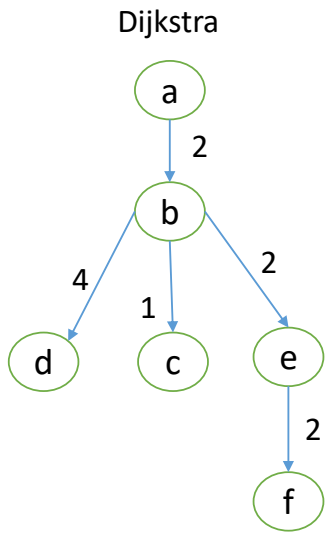


Prim-Dijkstra algorithm

The greedy loop of Prim selects an edge of G with minimum cost which is connected to a node x already in the current spanning tree, irrelevant of how far x is from the root node. For example, Prim selects (f, d) because it is the shortest edge to reach d between edges (e, d) and (b, d) even thought the path to f from a is quite long.

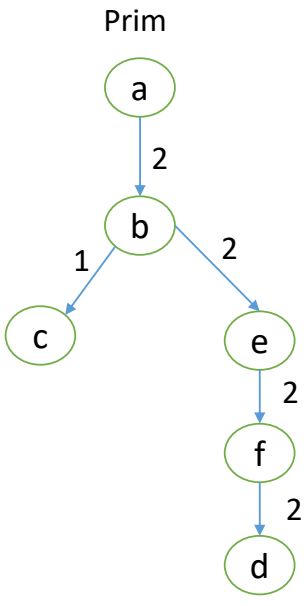


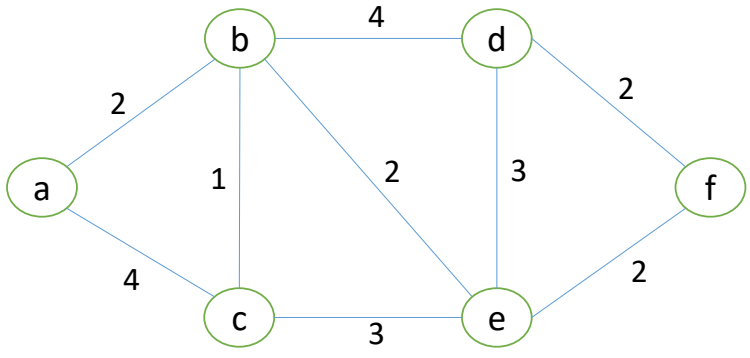


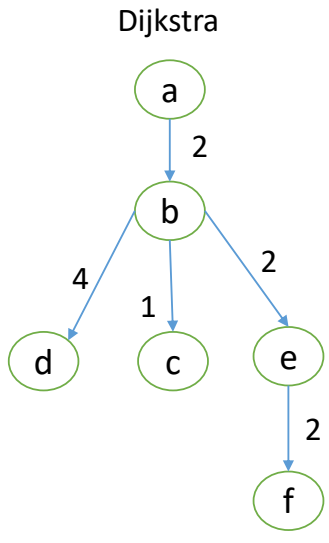


Prim-Dijkstra algorithm

The greedy loop of Dijkstra selects an edge of G that minimizes the distance of all existing paths leaving the source node (all existing paths means all the paths composed of nodes already selected). The edge (b, d) is selected last in the construction by Dijkstra of the tree, and this edge is selected rather than (e, d) or (f, d) because it minimizes the distance of the path from source node a to node d.







Prim-Dijkstra algorithm : notes

Prim and Kruskal only works for undirected graphs

Dijkstra works with both directed and undirected graphs. Directed graphs can be transformed into undirected graphs by adding for every arc (x, y) with cost $w(x, y) = z$ a new arc (y, x) with cost $w(y, x) = z$.

In the previous example, we assume that each edge (x, y) is actually two arcs in opposite directions of same costs between nodes x and y.

Encoding data : Huffman algorithm

Alphabetic characters used to be stored in *ASCII* on computers, which requires 7 bits per character.

ASCII is a **fixed-length code**, since each character requires the same number of bits to store.

Encoding data : Huffman algorithm

Notice that some characters, (e.g. **q**, **x**, **z**, **v**) are rare, and others (e.g. **e**, **s**, **t**, **a**) are common.

It might make more sense to use less bits to store the common characters, and more bits to store the rare characters.

An encoding that does this is called a variable-length code.

A code is called optimal if the space required to store data with the given distribution is a minimum.

Optimal codes are important for many applications.

Variable length code example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

Thus, the string *treat* is encoded as *110111011*

The problem with this encoding is that the string could also be *ktve*.

Variable length code example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

With this code, we have to somehow keep track of which letter is which.
(e.g. *11_01_1_10_11*)

To do this requires more space, and may make the code worse than a fixed-length code. Rather we use a prefix code.

Prefix codes

A code in which no word is a prefix of another word.

To encode a string of data, concatenate the codewords together.

To decode, just read the bits until a codeword is recognized.

Since no codeword is a prefix of another, this works.

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

Notice that no codeword is the prefix of another.

Now, we can encode *treat* as *1001010100100*, and it is uniquely decodable.

Decoding prefix codes

Decode *01111011010010100100* based on the following decoding table :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

The most obvious way is to read one bit, see if it is a character, read another, see if the two are a character, etc. : *not very efficient*.

Decoding prefix codes

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

A better way is to represent the encoding of each letter as a path in a binary tree :

- Each leaf node stores a character.
- A '0' means go to the left child
- A '1' means go to the right child
- The process continues until a leaf is found.

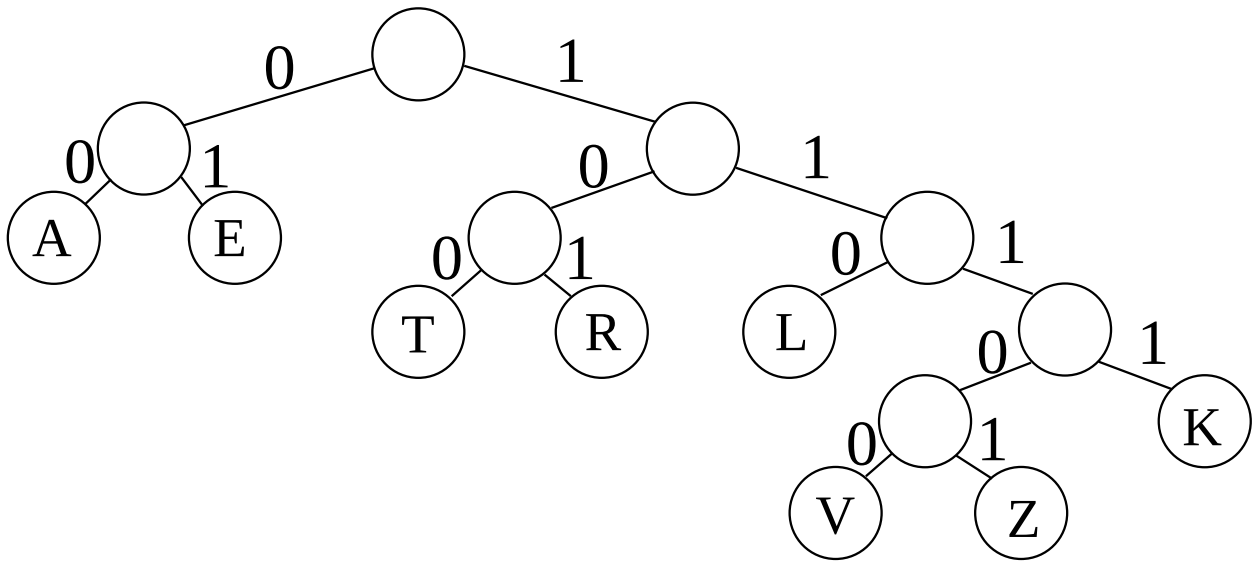
Any code represented this way is a prefix code. Why ?

Decoding prefix code

Decode *01111011010010100100*

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

We will represent the data by the following binary tree :



It is now not too hard to see that the answer is *ezrarat*.

Constructing codes

Prefix codes make encoding and decoding data very easy.

It can be shown that optimal data compression using a character code can be obtained using a prefix code.

How can we construct such a code ?

Huffman code is an algorithm to construct prefix codes

Huffman code

A Huffman code can be constructed by building the encoding tree from the bottom-up in a greedy fashion.

Since the less frequent nodes should end up near the bottom of the tree, it makes sense that we should consider these first.

We'll see an example, then the algorithm.

Huffman code example

Suppose I want to store this very long word in an electronic dictionary : *floccinaucinihilipilification*.

I want to store it using as few bits as possible.

The frequency of letters, sorted in decreasing order, is as follows :

i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

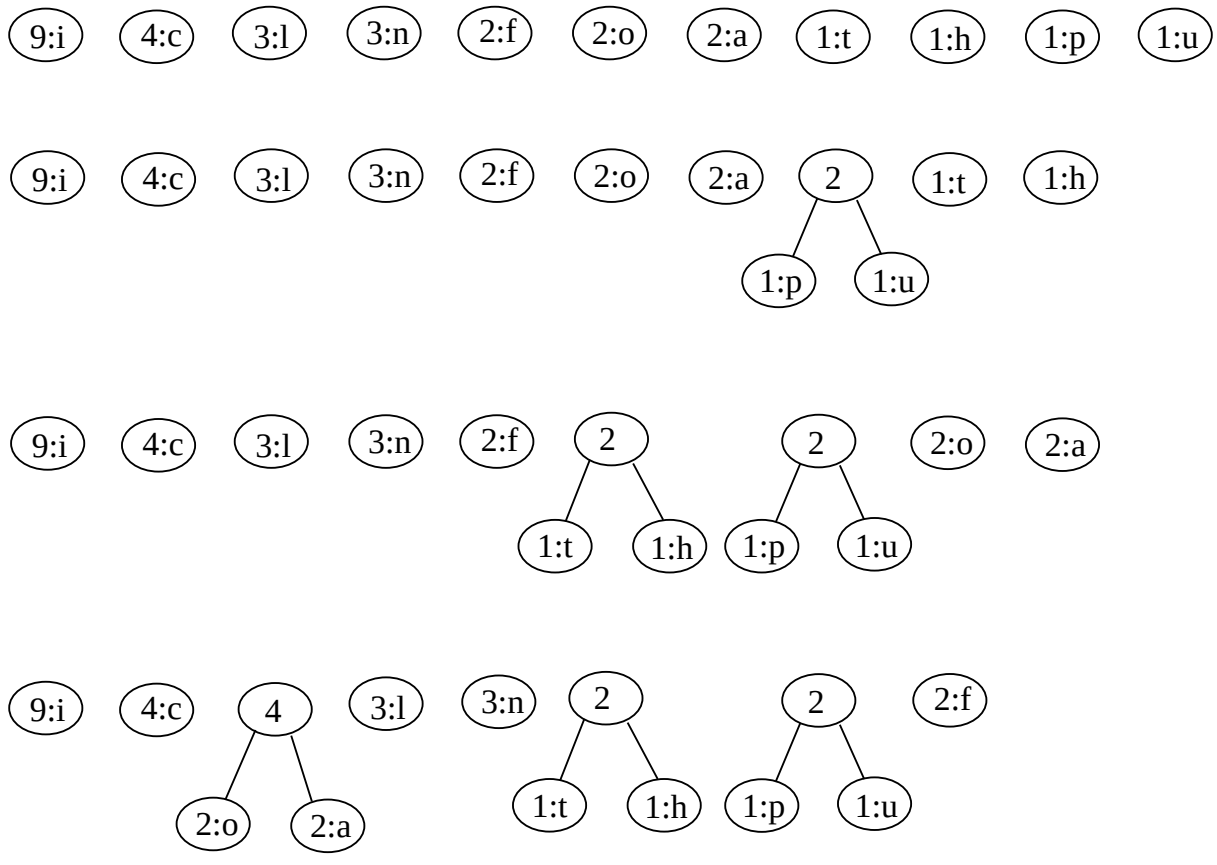
Huffman code example

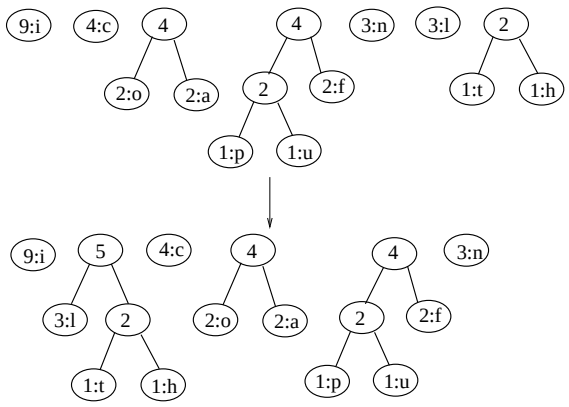
i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

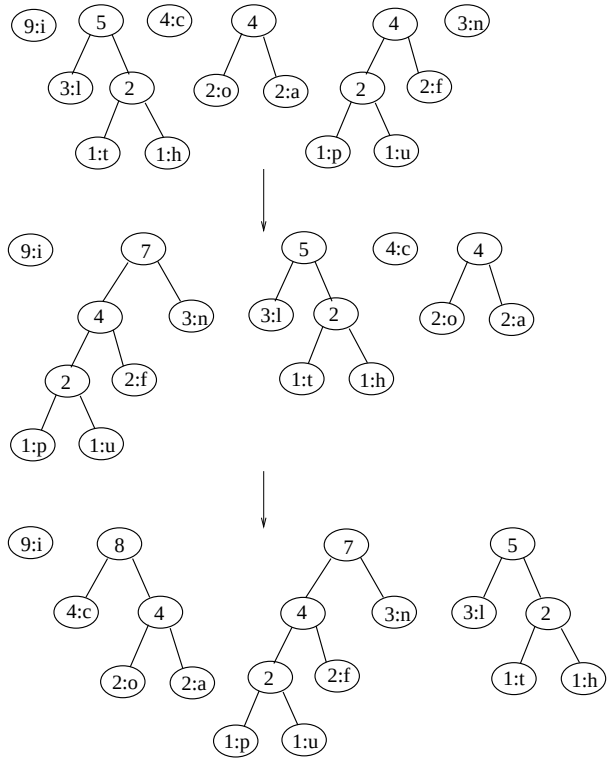
The algorithm works sort of like this :

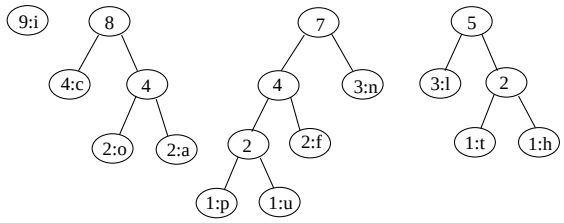
- Consider each letter as a node in a not-yet-constructed tree.
- Label each node with its letter and frequency.
- Pick two nodes x and y of least frequency.
- Insert a new node, and let x and y be its children. Let its frequency be the combined frequency of x and y.
- Take x and y off the list.
- Continue until only 1 node is left on the list.

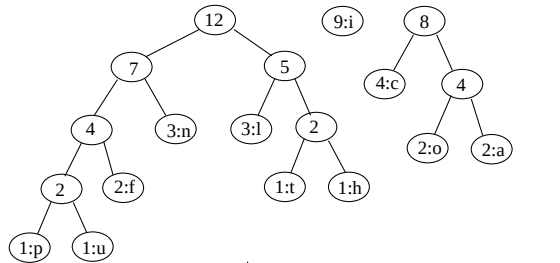
Huffman example

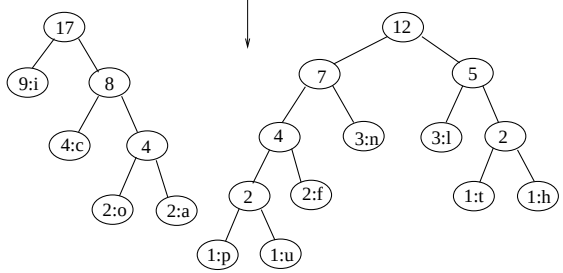


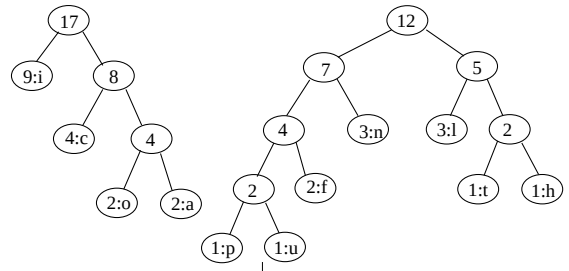

9:i
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n
3:l
2
1:t
1:h
9:i
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n

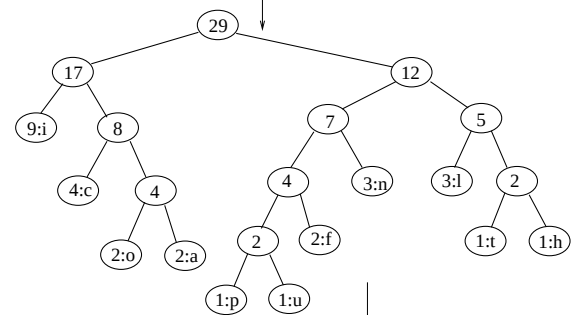

9:i
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n
9:i
7
4
2
2:f
1:p
1:u
3:n
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
9:i
8
4:c
4
2:o
2:a
7
4
2
2:f
1:p
1:u
3:n
5
3:l
2
1:t
1:h

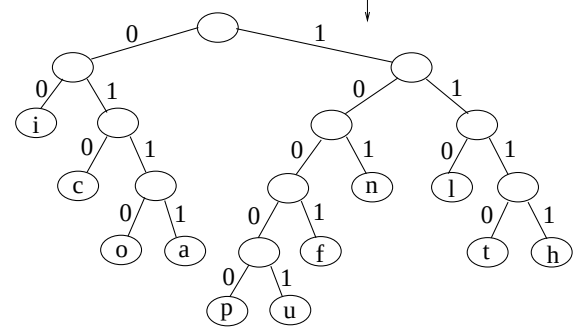

9:i
8
4:c
4
2:o
2:a
7
4
3:n
2
2:f
1:p
1:u
5
3:l
2
1:t
1:h


12
7
5
4
3:n
3:l
2
2:f
1:t
1:h
2
1:p
1:u
9:i
8
4:c
4
2:o
2:a

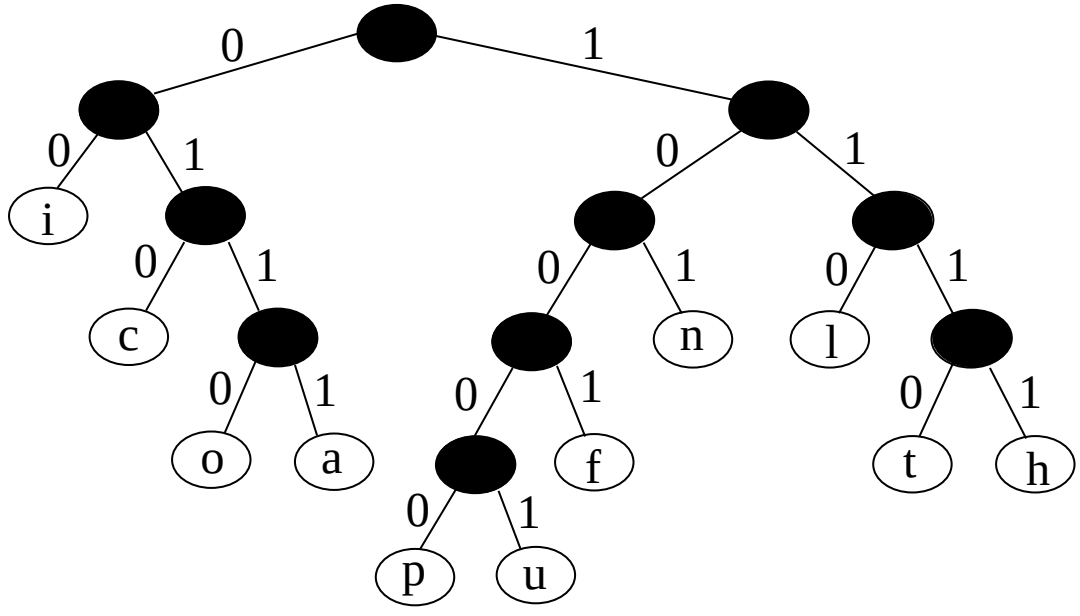

17
9:i
8
4:c
4
2:o
2:a
12
7
5
4
3:n
3:l
2
2
2:f
1:t
1:h
1:p
1:u


17
9:i
8
4:c
4
2:o
2:a
12
7
5
4
3:n
3:l
2
2
2:f
1:t
1:h
1:p
1:u


29
17
9:i
8
4:c
4
2:o
2:a
12
7
5
4
3:n
3:l
2
2
2:f
1:t
1:h
1:p
1:u


0
1
0
i
1
0
c
1
0
o
1
a
0
1
0
n
1
0
f
1
0
p
1
u
0
l
1
0
t
1
h

We can now list the code :



i (9)	00	f (2)	1001
c (4)	010	t (1)	1110
n (3)	101	h (1)	1111
l (3)	110	p (1)	10000
o (2)	0110	u (1)	10001
a (2)	0111		

Huffman encoding

We are given a list of n characters, each with some frequency.

For each character, we define a **Node** containing the *character*, *frequency*, *left*, and *right*.

The nodes are stored using a data structure that supports the operations **Insert** and **Extract_Min**.

A **priority queue**, which is implemented using a heap, is a good choice.

The algorithm for Huffman encoding will build a tree from the nodes in a bottom-up fashion.

Huffman encoding algorithm & complexity analysis

```
Huffman_Encoding(The_List)
  Q = The_List;
  /*Initialize the min priority queue */
  while (Q > 1)
    z := New Tree_Node;
    x := Extract_Min(Q);
    y := Extract_Min(Q);
    left[z] := x;
    right[z] := y;
    f[z] := f[x] + f[y];
    Insert(Q,z);
  return Extract_Min(Q);
```

The The_List is a n characters list each character with its own frequency. Q is a min priority queue key on the frequency of the characters

The most costly operations are Q = The_List which takes $O(n)$ to build the heap, **Insert** and **Extract_Min** operations which run in $O(\log n)$. The **while** loop iterates $n-1$ times, therefore the algorithm runs in $O(n \log n)$.

Huffman code is optimal

Let C be the set of character in a file.

We can imagine that the Huffman algorithm combines two characters x and y into a new character z with frequency $f(x) + f(y)$, and tries to find an optimal prefix code for $C' = C \cup \{z\} - \{x, y\}$.

Huffman code is optimal

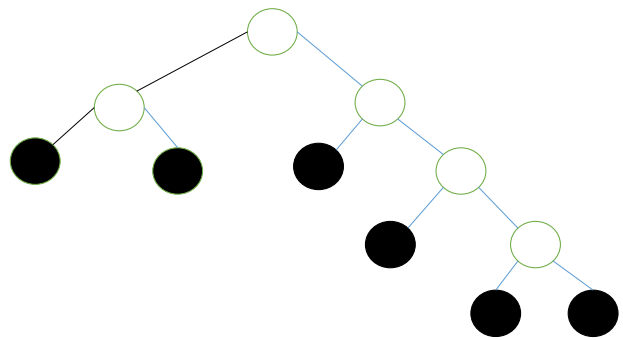
To prove that the Huffman code is optimal, we would need to show two things :

- Making the greedy choice will result in an optimal prefix code. In other words, if x and y are characters of minimum frequency, then there exists some optimal prefix code in which x and y are siblings. (**greedy choice property**)
- If T is an optimal code for C containing siblings x and y with parent z, then $T - \{x, y\}$ is an optimal code for $C' = C \cup \{z\} - \{x, y\}$. (**optimal substructure**)

We will not prove these here.

Exercise on Huffman code

1. You are given the following table of letters and their frequencies : a :1 b :3 c :2 d :9 e :7
   - Build the Huffman tree for this set of letters
   - Give the optimal Huffman code for these letters
2. What is the optimal Huffman code for the following set of frequencies : a :1 b :1 c :2 d :3 e :5 f :8 g :13 h :21
3. Write a frequency list of the Huffman code that creates the following structure



Greedy algorithm : first conclusions

- An optimization problem can be solved efficiently using a greedy algorithm if it exhibits the following properties :
  - **greedy choice property**
  - **optimal substructure**
- Finding a greedy algorithm may not be too difficult, however proving that it always returns the optimal solution is often challenging
- Greedy algorithms are computationally efficient because, unlike dynamic programming, they don't solve every optimal subproblem.
- In some cases, greedy algorithms can be used to produce sub-optimal solutions (greedy heuristics). That is, solutions which aren't necessarily optimal, but are perhaps very close.

Greedy algorithms : conclusion, greedy heuristics

Many problems have no known efficient algorithms (i.e. algorithms that run in polynomial time).

For these problems we know that there exist no greedy algorithm (obviously)

However the greedy choice of greedy algorithms can be used to design sub-optimal solutions

Greedy algorithms : conclusion, greedy heuristics

An example is the traveling salesman problem which is an undirected weighted graph for which we have to visit every node exactly once while minimizing the distance. Here is a greedy heuristic based on Kruskal's algorithm :

As in Kruskal's algorithm, first sort the edges in the increasing order of weights. Starting with the least cost edge, look at the edges one by one and select an edge only if the edge, together with already selected edges,

1. does not cause a vertex to have degree three or more
2. does not form a cycle, unless the number of selected edges equals the number of vertices in the graph

Exercises on Huffman code

1. What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers ? a :1 b :1 c :2 d :3 e :5 f :8 g :13 h :21
2. Encode the following sentence with a Huffman code : "Common sense is the collection of prejudices acquired by age eighteen". Write the complete construction of the code
3. Write a minimal character-based binary code for the following sentence : "in theory, there is no difference between theory and practice ; in practice, there is"
   The code must map each character, including spaces and punctuation marks, to a binary string so that the total length of the encoded sentence is minimal. Use a Huffman code and show the derivation of the code.

Problem 1 : The Job Sequencing Problem

We are given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline

```
Input: Four Jobs with following deadlines and profits
  JobID    Deadline      Profit
    a         4             20
    b         1             10
    c         1             40
    d         1             30
```

Every job takes a single unit of time, so the minimum possible deadline for any job is 1. The objective is to maximize the total profit when only one job can be scheduled at a time

Design a greedy algorithm to solve this problem

The Job Sequencing Problem

Using your greedy algorithm, solve the following job sequencing problem

```
Input:  Five Jobs with following deadlines and profits
   JobID      Deadline      Profit
     a           2            100
     b           1             19
     c           2             27
     d           1             25
     e           3             15
```

problem 2 : Activity scheduling problem

Assume we have a set S of n activities with each of them being represented by a start time s_i and finish time f_i :

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Two activities i and j are said to be non-conflicting if $s_i \geq f_j$ or $s_j \geq f_i$ (the two activities do not overlap).

The optimization problem is to maximize the number of non-conflicting activities. Application : select the maximum number of activities that can be performed by a single person or machine.

Design a greedy algorithm to solve this problem

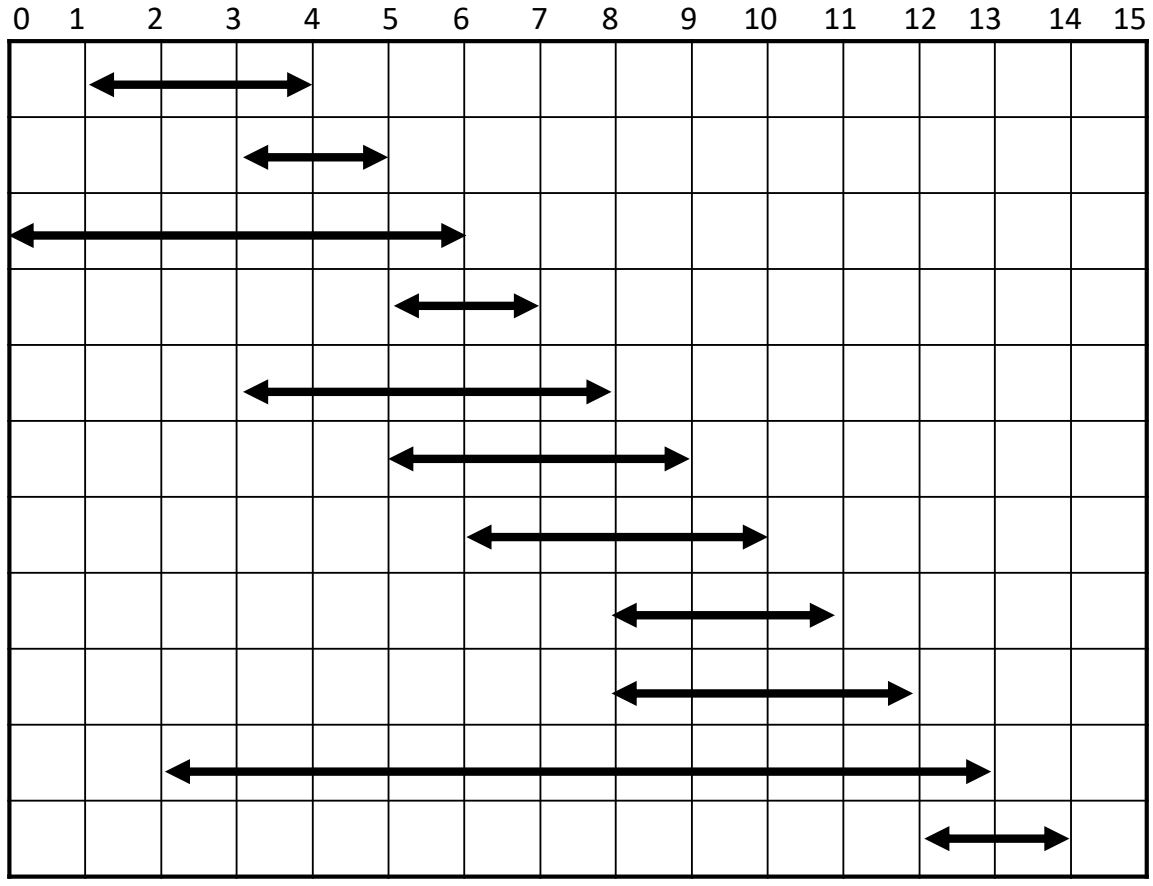
Activity scheduling problem : a greedy algorithm

Greedy algo : Sort the activities in increasing order of their finish time (greedy criterion). Then :

- Select the activity with the earliest finish
- Eliminate the activities that overlap with the selected activity
- Repeat !

Activity scheduling problem

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Activity scheduling problem : greedy choice property

Let $S = \{1, 2, \ldots, n\}$ be the set of activities ordered by finish time.

Assume that $A \subseteq S$ is an optimal solution, also ordered by finish time.

Assume the index of the first activity in A is $k \neq 1$, i.e., this optimal solution does not start with the greedy choice.

We show that $B = (A \setminus \{k\}) \cup \{1\}$ which begins with the greedy choice (activity 1), is another optimal solution.

Since $f_1 \leq f_k$, and the activities in A are disjoint by definition, the activities in B are also disjoint. Since B has the same number of activities as A, that is $|A| = |B|$, then B is also optimal.

Activity scheduling problem : optimal substructure

Once the greedy choice is made, the problem reduces to finding an optimal solution for a subproblem.

If A is an optimal solution to the original problem S, then $A' = A \setminus \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : s_i \geq f_1\}$.

Otherwise, if we could find a solution B' to S' with more activities then A', adding 1 to B' would yield a solution B to S with more activities than A, contradicting the optimality assumption of A.

Problem 3 : minimize average completion time

This is a scheduling problem which consists to minimize the average completion time of tasks.

Given a set $S = \{t_1, t_2, \ldots, t_n\}$ of tasks, where task t_i requires p_i units of processing time to complete, once it has started. There is one computer on which to run these tasks, and the computer can run only one task at a time. Let f_i be the completion time of task t_i, that is, the time at which task t_i completes processing. The optimization problem is to minimize the average completion time, i.e. minimize $\frac{\sum_{i=1}^{n} f_i}{n}$.

Example : two tasks, t_1 and t_2, $p_1 = 3$ and $p_2 = 5$. Assume t_2 runs first, followed by t_1. Then $f_2 = 5$, $f_1 = 8$, and the average completion time is $\frac{5+8}{2} = 6.5$. If task t_1 runs first, then $f_1 = 3$, $f_2 = 8$, and the average completion time is $\frac{3+8}{2} = 5.5$

Give a greedy algorithm that schedules the tasks to minimize the average completion time. Each task runs non-preemptively, once task t_i starts, it must run continuously for p_i units of time.