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

21 juin 2021

Outline

Greedy algorithms

Greedy algorithms can be used to solve the same type of ”optimization” problems where divide&conquer or dynamic programming are applied, such as making change, 0-1 knapsack or shortest path problems

In all theses problems the solution consist to select a subset of objects optimizing some objective function :

- Making change : select the minimum subset of coins to return change
- Knapsack : select a subset of objects that maximize profit

Greedy algorithms

Finding the right subsets of object is complicated using divide&conquer or dynamic programming, while solutions by greedy algorithms are extremely simple :

- order the object according to some "desirability" (greedy) criterion
- select the objects based on the "greedy criterion" used to order them

For example, objects of a 0-1 knapsack can be sorted based on their value, object with the greatest value come first

Then objects are put in the knapsack according to this greedy criterion if their weight is smaller than the residual capacity of the knapsack

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_objects[n] = {0} ;
int objects[n] ;
int i, j, w = W
Sort objects[] according either to their values, weights or density $\frac{v[i]}{w[i]}$
for (i=1 ; i $\leq n$, i++)
   j = objects[i]
   if ($w > w_j$)
      selected_objects[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 $selected\_objects[n] = \{0\}$ ; int $w = W$ ; int $objects[n]$ ; int $i, j$
Sort in $objects[n]$ objects along values
for $(i = 1; i \leq n, i++)$
  $j = objects[i]$
  if $(w > w_j)$
    $selected\_objects[i] = j$
    $w = w - w_j$
  $i++$
```

$objects = [3, 4, 5, 1, 2]$ after sorting

$i = 1$, $object = 3$, $selected\_objects = [3, 0, 0, 0, 0]$, $w = 6$

$i = 2$, $object = 4$, $selected\_objects = [3, 4, 0, 0, 0]$, $w = 0$

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 $selected\_objects[n] = \{0\}$ ; int $w = W$ ; int $objects[n]$ ; int $i, j$
Sort in $objects[n]$ objects along weights
for $(i = 1; i \leq n, i++)$
  $j = objects[i]$
  if $(w > w_j)$
    $selected\_objects[i] = j$
    $w = w - w_j$
  $i++$
```

$objects = [1, 2, 5, 3, 4]$ after sorting

$i = 1$, $object = 1$, $selected\_objects = [1, 0, 0, 0, 0]$, $w = 9$

$i = 2$, $object = 2$, $selected\_objects = [1, 2, 0, 0, 0]$, $w = 6$

$i = 3$, $object = 5$, $selected\_objects = [1, 2, 5, 0, 0]$, $w = 2$

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

$objects = [1, 3, 5, 4, 2]$ after sorting

$i = 1$, $object = 1$, $selected\_objects = [1, 0, 0, 0, 0]$, $w = 9$

$i = 2$, $object = 3$, $selected\_objects = [1, 3, 0, 0, 0]$, $w = 4$

$i = 3$, $object = 5$, $selected\_objects = [1, 2, 5, 0, 0]$, $w = 0$

Solution is 25

Problems where greedy algorithms fail

We have just see that greedy algorithm for the same 0-1 knapsack problem instance returns 3 different solutions !

This because greedy algorithms may fail to solve some optimization problems to optimality, this is the case for 0-1 knapsack and the making change problem

There are other optimization problems for which greedy work perfectly well, in this case greedy is a wonderful algorithm because it is simple and computationally very efficient (low asymptotic complexity)

Greedy works when a problem satisfies a second condition (beside the optimal substructure condition) : the greedy choice condition

We will not have time to elaborate on the greedy choice condition.

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.

Usually there are several spanning trees for a same graph

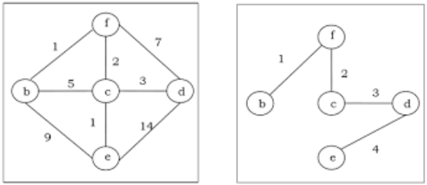


Figure – Weighted graph and corresponding spanning tree

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.

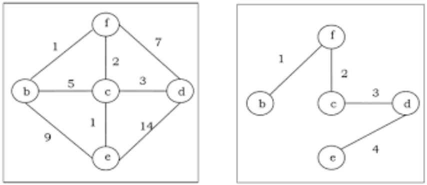
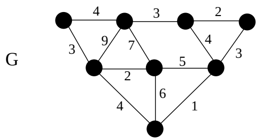
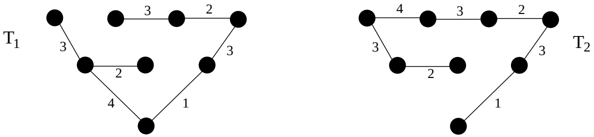


Figure – Weighted graph and corresponding spanning tree

Minimum Spanning Tree : Examples

There could also be more than one MST for a same graph. In this example, there are two minimum spanning trees.





Constructing an MST

The 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

Initially makes each node of the graph as a tree with a single node, i.e. a forest of trees

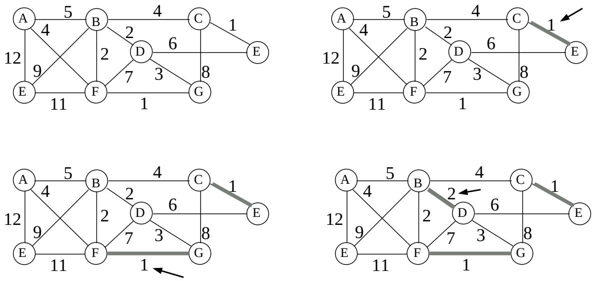
Then greedily select the shortest edge no making a cycle to be part of the MST

This edge merge two trees into a single one

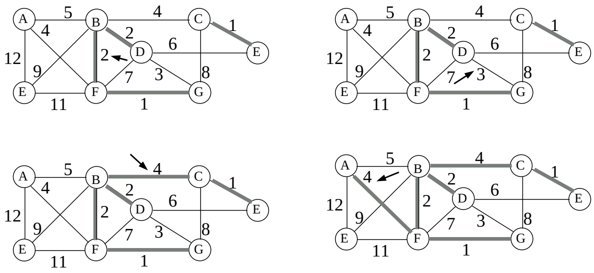
```
Algorithm Kruskal($G$)
input : Graph $G = (V, E)$;
output : A minimum spanning tree $T$;
  Sort $E$ by increasing length;
  $n = |V|$;
  $T = \emptyset$;
  For each $v \in V$ MakeSet(v)
  for $i = 1$ to $|E|$ do /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered;
    if (FindSet(u)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

Kruskal's Algorithm Example

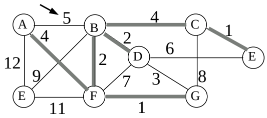
```
for $i = 1$ to $|E|$ do /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered ;
    if (FindSet(u) $\neq$ FindSet(v)) then
        Union(FindSet(u), FindSet(v))
        $T = T \bigcup \{u, v\}$ ;
return $T$
```

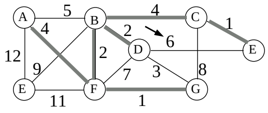


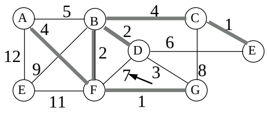
```
for $i = 1$ to $|E|$ do /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered ;
    if (FindSet(u) $\neq$ FindSet(v)) then
        Union(FindSet(u), FindSet(v))
        $T = T \bigcup \{u, v\}$ ;
return $T$
```

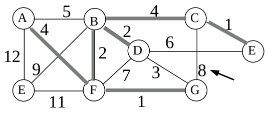


```
for $i = 1$ to $|E|$ do /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered ;
    if (FindSet(u) $\neq$ FindSet(v)) then
        Union(FindSet(u), FindSet(v))
        $T = T \bigcup \{u, v\}$ ;
return $T$
```

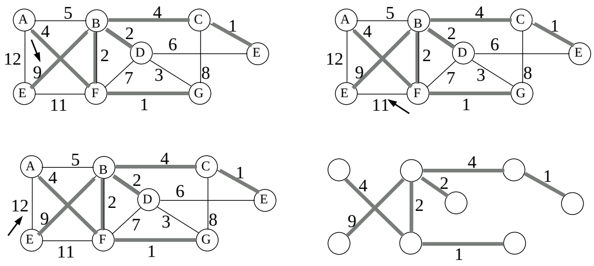








```
for $i = 1$ to $|E|$ do /* Greedy loop */
    $\{u, v\}$ = shortest edge not yet considered ;
    if (FindSet(u) $\neq$ FindSet(v)) then
        Union(FindSet(u), FindSet(v))
        $T = T \bigcup \{u, v\}$ ;
return $T$
```



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

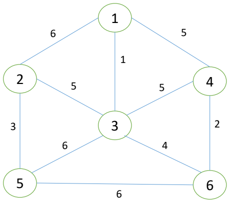
Consider the non-oriented weighted graph *G* below represented using an adjacency matrix

	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.

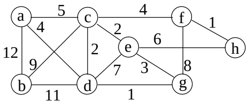


Prim's Algorithm

1. Prim selects arbitrary a node A of the graph to be the root of the MST T
2. Then selects the shortest edge adjacent to A to be the first edge of T
3. Prim grows the MST T by selecting the shortest edge adjacent to a node of T, not yet in T and not forming a cycle

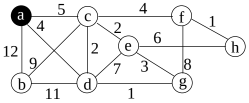
```
Prim_MST($G, r$)
  $\forall\ u \in G$
    $key[u] =$ Max_Int ;
  $key[r] = 0$ ;
  $p[r] =$ NIL ;
  $Q =$ MinPriorityQueue($V$)
  **while** ($Q \neq \emptyset$)
    $u =$ ExtractMin($Q$) ;
    **for** each $v$ adjacent to $u$
      **if** (($v \in Q$) & ($w(u,v) < key[v]$))
        $p[v] = u$ ;
        $key[v] = w(u,v)$ ;
```

```
while ($Q \neq \emptyset$)
  $u$ = ExtractMin($Q$) ;
  for each $v$ adjacent to $u$
    if (($v \in Q$)
    & ($w(u,v) < key[v]$))
      $p[v] = u$ ;
      $key[v] = w(u,v)$ ;
```



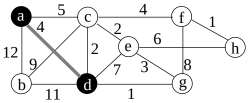
v	a	b	c	d	e	f	g	h
k	0	∞	∞	∞	∞	∞	∞	∞
p	nil	?	?	?	?	?	?	?

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



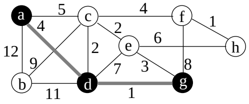
v	a	b	c	d	e	f	g	h
k	~~0~~	12	5	4	∞	∞	∞	∞
p	nil	a	a	a	?	?	?	?

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	?

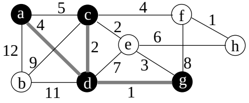
Q=[g,c,e,b,f,h]



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	?

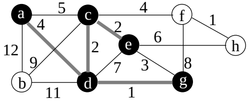
Q=[c,e,f,b,h]

```
while ($Q \neq \emptyset$)
  $u = \text{ExtractMin}(Q)$ ;
  for each $v$ adjacent to $u$
    if (($v \in Q$)
    & ($w(u,v) < key[v]$))
      $p[v] = u$ ;
      $key[v] = w(u,v)$ ;
```



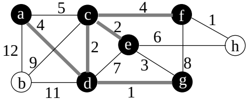
v	a	b	c	d	e	f	g	h
k	$\cancel{0}$	9	$\cancel{2}$	$\cancel{4}$	2	4	$\cancel{1}$	∞
p	nil	c	d	a	c	c	d	?

Q=[e,f,b,h]



v	a	b	c	d	e	f	g	h
k	$\cancel{0}$	9	$\cancel{2}$	$\cancel{4}$	$\cancel{2}$	4	$\cancel{1}$	6
p	nil	c	d	a	c	c	d	e

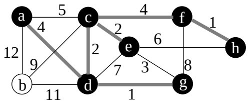
Q=[f,h,b]



v	a	b	c	d	e	f	g	h
k	$\cancel{0}$	9	$\cancel{2}$	$\cancel{4}$	$\cancel{2}$	$\cancel{4}$	$\cancel{1}$	1
p	nil	c	d	a	c	c	d	f

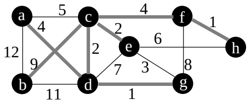
Q=[h,b]

```
while ($Q \neq \emptyset$)
  $u =$ ExtractMin($Q$) ;
  for each $v$ adjacent to $u$
    if (($v \in Q$)
    & ($w(u,v) < key[v]$))
      $p[v] = u$ ;
      $key[v] = w(u,v)$ ;
```



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

Q=[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=[]

Prim's Algorithm : time complexity analysis

```
Prim_MST($G, r$)
  $\forall\ u \in G$
    $key[u]$ = Max_Int ;
  $key[r] = 0$ ;
  $p[r]$ = NIL ;
  $Q$ = MinPriorityQueue($V$)
  while ($Q \neq \emptyset$)
    $u$ = ExtractMin($Q$) ;
    for each $v$ adjacent to $u$
      if (($v \in Q$) & ($w(u, v) < key[v]$))
        $p[v] = u$ ;
        $key[v] = w(u, v)$ ;
```

Priority Queue : $O(n)$

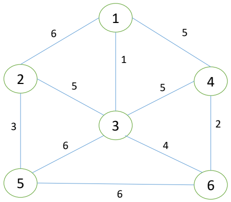
while loop runs n times

- 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

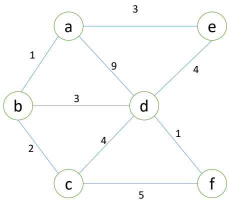
Compute the MST using Prim's algorithm. At each step of the computation, show the configuration of the partial MST.



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						

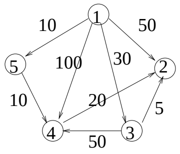


The single-source shortest paths problem

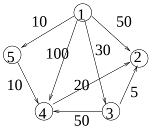
Let $G = \{V, E\}$ be a connected directed graph where V is the set of nodes and E is the set of arcs.

- 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



Greedy aspects of 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}

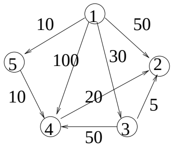
Initially, the path from s to any other node x is the direct edge (s, x)

These edges are sorted in increasing order of their length, greedy algo selects the shortest edge (s, y) among them, this become the first shortest path

Then Dijkstra's algo re-calculate the distance of the path from s to any node other node x (different from y) using the shortest path (s, y) and edge (y, x) if such an edge exist

Those paths are sorted in increasing order of their length, greedy algo (Dijkstra) selects the shortest one, this become the second shortest path

Greedy aspects of Dijkstra's algorithm

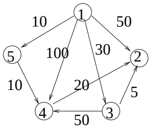


Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

In general, each time a new shortest path $(s, ..., y)$ has been found,

- Dijkstra's algo re-calculate the distance from the source node s to any other node x for which the shortest path $(s, ..., x)$ is yet to be found
- The re-calculation is obtained by creating a path $(s, ..., y, x)$, if this path is shortest than the existing path $(s, ...x)$, then the length of $(s, ..., y, x)$ becomes the lenght of the path from s to x.
- Then Dijkstra selects the shortest of all these paths that link s to a node x for which the shortest path has not yet be computed

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}

Here each time a new shortest path is computed, the distance from the source node to all the other nodes is re-calculated and stored into the distance vector D.

```
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 : obtaining the shortest paths

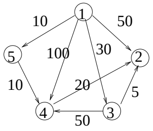
To obtain the shortest paths (not just their length), we need a new array $P[2..n]$ where $P[v]$ contains the node that precedes v in the shortest path

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

- $v = 3$

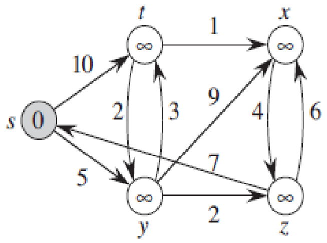
P	1	2	3	4	5
	1	3	1	5	1

- $v = 2$

P	1	2	3	4	5
	1	3	1	5	1

Exercise : single-source shortest paths

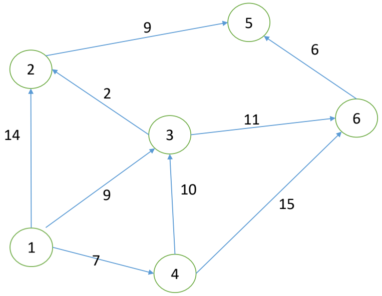
Compute the single-source shortest paths for the oriented graph below. The source is node s.



```
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 $D$
```

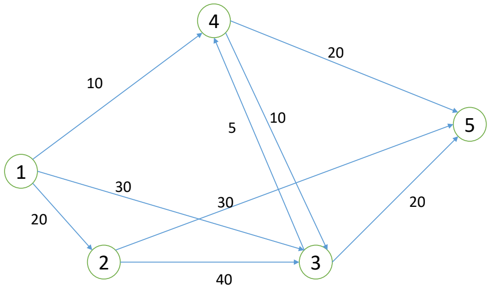
Exercise : single-source shortest paths

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.



Encoding data : Huffman algorithm

Huffman algorithm build prefix codes that are optimal in terms of the number of bits needed to encode a text

The Huffman algorithm is a greedy algorithm, it is another example of optimization problem that can be solved optimally using a greedy algorithm

Fixed-length codes

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.

Variable length codes

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.

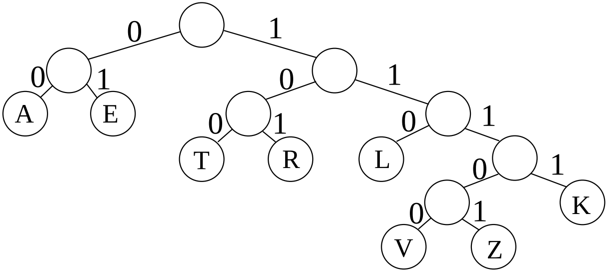
Any code represented this way is a prefix code. Why ?

Decoding prefix code

Decode *0111101101001010010O*

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

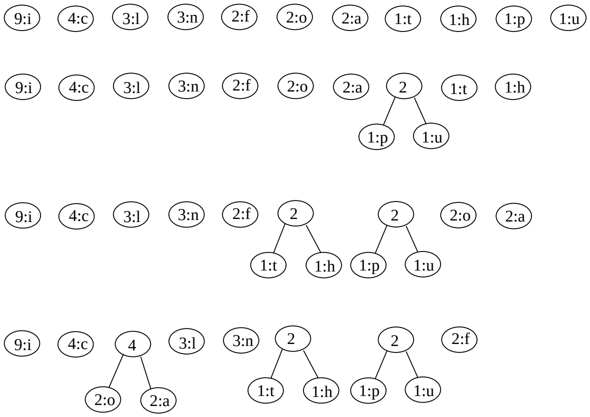
Huffman code example

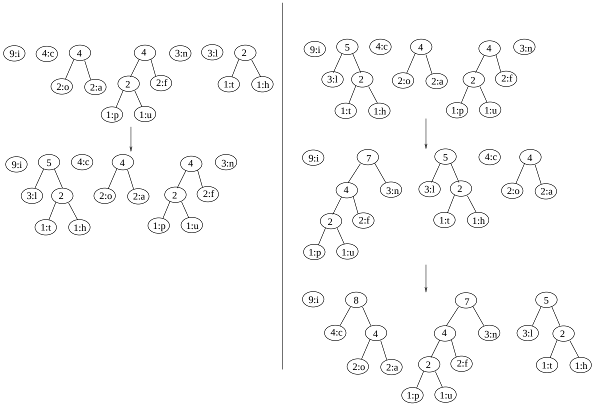
i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

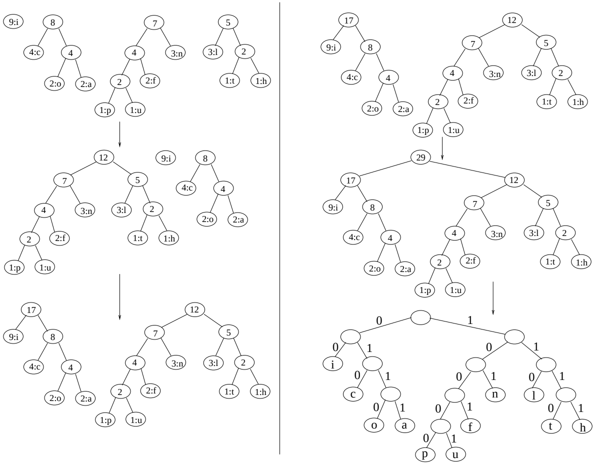
The algorithm works sort of like this :

- 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

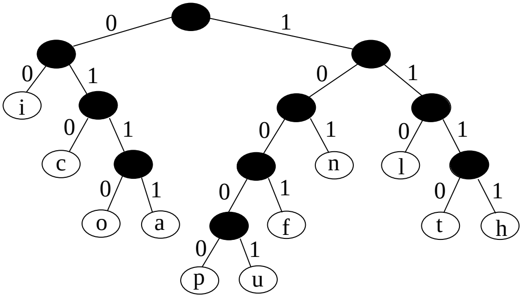



9:i
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n
3:l
2
1:t
1:h
9:i
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n
9:i
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
4
2
2:f
1:p
1:u
3:n
9:i
7
4
3:n
2
2:f
1:p
1:u
5
3:l
2
1:t
1:h
4:c
4
2:o
2:a
9:i
8
4:c
4
2:o
2:a
7
4
3:n
2
2:f
1:p
1:u
5
3:l
2
1:t
1:h



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
17
29
0
1
i
c
o
a
p
u
f
n
l
t
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

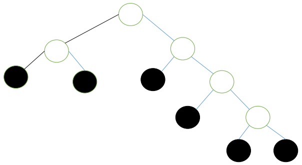
```
Huffman_Encoding(The_List)
  Q = The_List ;
  /*Initialize the min priority queue */
  while (Q $> 1$)
    $z :=$ New Tree_Node ;
    $x :=$ Extract_Min(Q) ;
    $y :=$ Extract_Min(Q) ;
    left$[z] := x$ ;
    right$[z] := y$ ;
    $f[z] := f[x] + f[y]$ ;
    Insert(Q,z) ;
  return Extract_Min(Q)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

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 $O(n \log n)$.

Exercises 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



Exercises on Huffman code

4. 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
5. 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

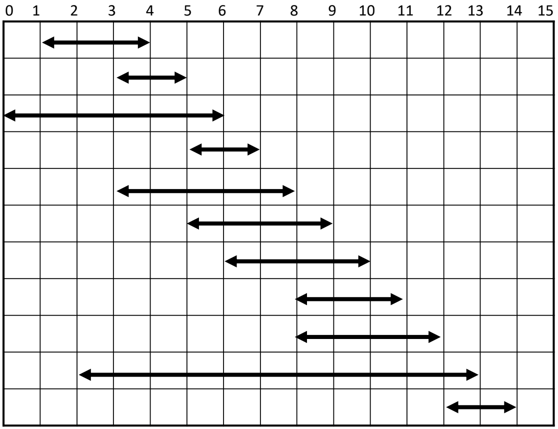
Activity scheduling problem : a greedy algorithm

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.