Greedy Algorithms I [MIT Open Courseware 6.046, Chapter 16 - Greedy Algorithms, Chapter 23 - Cormen 2001]

- Lets start by reviewing some theory about graphs
- Direct Graph = Digraph $G = (V, E)$
  - Set V of vertices
  - Set $\underbrace{E \subseteq V \times V}_{E \text{ is a subset of the product of } V \times V}$ of edges
- Undirected Graph: E contains unordered pairs
- $|E| = O(|V|^2)$ = "The number of edges is at most $|V|^2$"
- If G is connected, i.e. there is a path from any vertex to any other vertex: $|E| \geq |V| - 1$
- Accordingly, if G is connected: $\boxed{|E| = O(|V|^2)}$ $\boxed{|E| \geq |V| - 1}$

This implies:

- $|E| = \boxed{O(|V|^2)}$, $|E| = \boxed{\Omega(|V|)}$ } This is not very useful. What if we try to use logarithms instead?
- By taking logarithms:

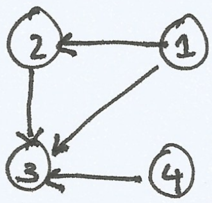
$$\left.\begin{array}{l} \log|E| = O(\log|V|^2) = O(2\log|V|) = O(\log|V|) \\ \log|E| = \Omega(\log|V|) \end{array}\right] > \Theta(\log|V|)$$

– Graph Representation:

- Adjacency Matrix of $G = (V, E)$ where $V = \{1, 2, \dots, n\}$ is the $n \times n$ matrix $\underline{A}$ given by:

$$A[i,j] = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{if } (i,j) \notin E \end{cases}$$

Example:



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Adjacency Matrix

e.g. complete graph (every vertex is connected to all vertices)

Requires $\Theta(|V|^2)$ storage $\Rightarrow$ dense representation

$\Downarrow$

works well if $|E| \approx |V|^2$ (i.e. if the number of edges is close to the maximum possible number of edges)

$\Leftarrow$

But for many graphs $|E| \ll |V|^2$ (i.e. the number of edges is much smaller than the maximum possible number of edges

$\Downarrow$

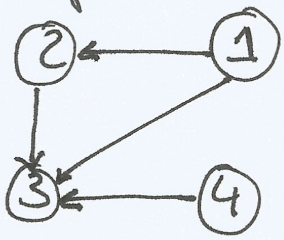
In this case the graph is said to be sparse $\Rightarrow$

Sparse graph examples:
- Linked list
- Doubly linked list
- Any tree

- For sparse graphs it is undesirable to spend $\Theta(|V|^2)$ memory. In those cases we can use an adjacency list

- **Adjacency List** of $v \in V$ is the list Adj[v] of vertices adjacent to v

Example:



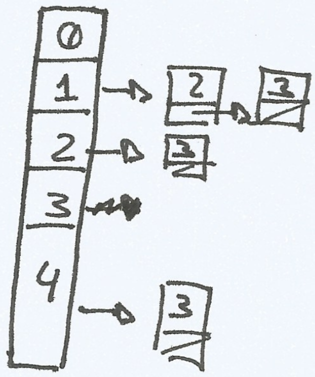
$Adj[1] = \{2, 3\}$

$Adj[2] = \{3\}$

$Adj[3] = \{\}$

$Adj[4] = \{3\}$

In computational array form



The length of the adjacency matrix of vertex v, i.e. $|adj[v]|$, is the number of connections from v to all other vertexes. This is the degree of vertex v. I.e.:

$$|Adj[v]| = \begin{cases} \text{degree}(v) & \text{if undirected graph} \\ \text{out-degree} & \text{if directed graph} \end{cases}$$

- An important lemma:

<u>Handshaking lemma</u> (undirected graph)

- $\sum_{v \in V} \text{degree}(v) = 2|E|$

- Every time we add one edge we increase by 1 the degree of each vertex

- For undirected graphs this implies that the adjacency list representation uses $O(2E)^{=O(E)}$ or more precisely $\Theta(V + E)$ storage

V adjacency lists each with $\Theta(E)$ elements to

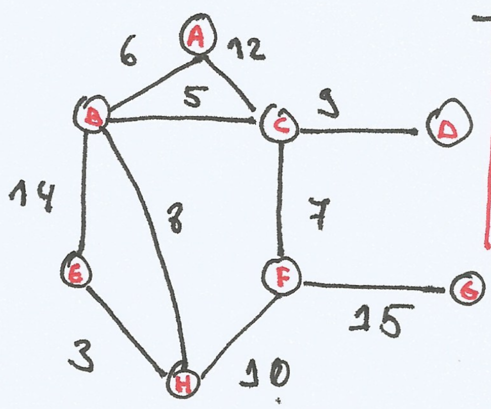
- Now lets proceed to the greedy algorithms by focusing on MSTs

Minimum Spanning Trees

- One of the world's most important algorithms
- Important in distributed systems
- Huge number of applications
- **Problem:**
  - Input: Connected, undirected graph $G = (V, E)$ with an edge weight function $w: E \to \mathbb{R}$
    - For simplicity, assume all edges weights are distinct
    - This implies that w is injective (1-to-1)
  - Output: A spanning tree T (i.e. connects all vertices without any cycles) of minimum weight.

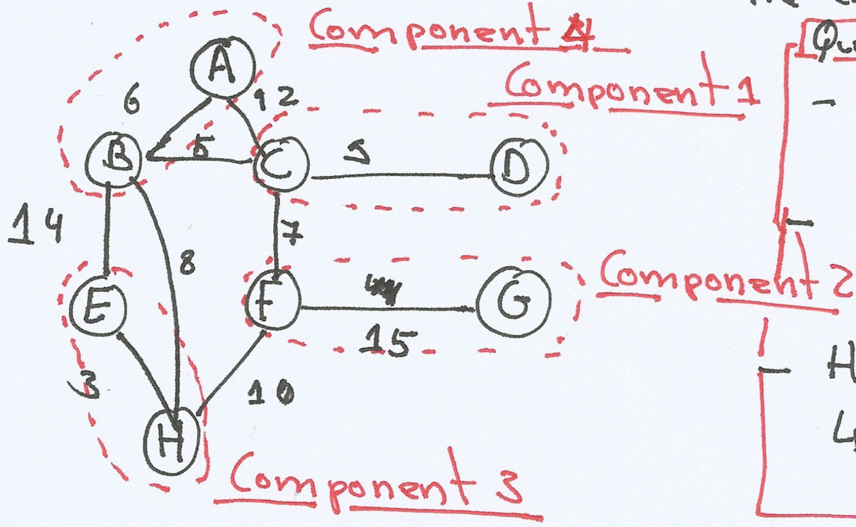
$$w(T) = \sum_{(v,u) \in T} w(v, u)$$

- Example:



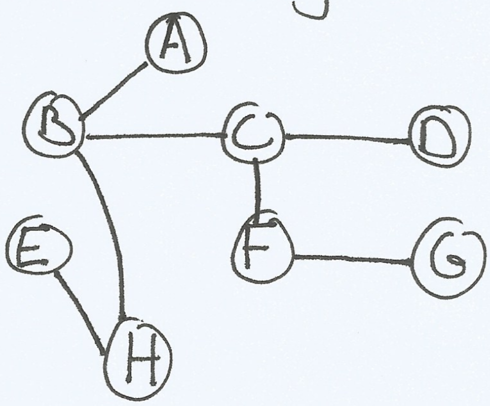
Objective:
- We want to find a tree (i.e. a connected acyclic graph) such that every vertex is part of the tree but it has to have the minimum weight possible
- The Edges with weight 9 and 15 have to be included since they are the only ones connecting ~~those~~ the nodes C ↔ D, F ↔ G

- The edge with weight 3 also has to be included since it is the minimum of edges connecting to node E
- The edge with weight 6 (B↔A) also has to be included since it is the minimum of the edges connecting to A. The other edge, respectively C↔A has weight 12
- The edges with weights 5 (B↔C), 8 (B↔H) and 7 (C↔F) also need to be included since they connect the previous components of the MST in the cheapest way:

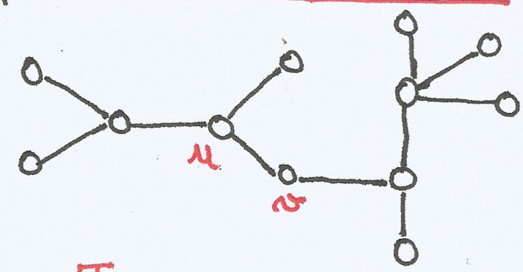


Questions:
- How can we connect C1 with C4
  ↳ Cheapest is through B↔C
- How can we connect C1 with C2?
  ↳ Cheapest is through C↔F
- How can we connect C3 with C4
  ↳ Cheapest is through B↔H

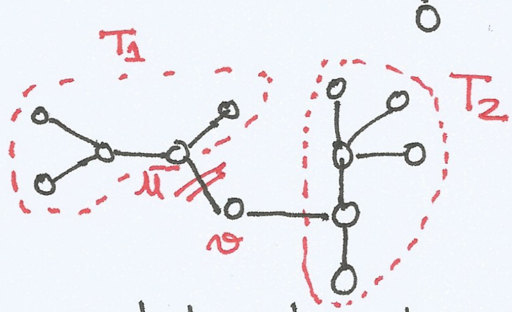
The resulting MST:



- MST have an <u>optimal substructure property</u>:
  - MST T: (other edges not shown)



  - Remove $(u,v) \in T$:



By the theorem below:
- T_1 is a MST of $G_1(V_1, E_1)$
- T_2 is a MST of $G_2(V_2, E_2)$

    - Then $\underline{T}$ is partitioned into two subtrees $\underline{T_1}$ and $\underline{T_2}$

  - <u>Theorem</u>: T_1 is a MST for $G_1 = (V_1, E_1)$ the subgraph of G induced by the vertices in T_1, i.e.:
    - V_1 = vertices in T_1
    - $E_1 = \{ (u,y) \in E_1 \mid u, y \in V_1 \}$

    Similarly for T_2

<u>Proof</u>:

- $W(T) = \underbrace{w(u,v)}_{\text{weight of the removed edge}} + w(T_1) + w(T_2)$
- Suppose that there was some T_1' that was $\underbrace{\text{better}}$ than T_1 for G_1 then we could make up a $T' = \{(u,v)\} \cup T_1' \cup T_2$ would be better than T for G

  better (lower weight) way to form a spanning tree

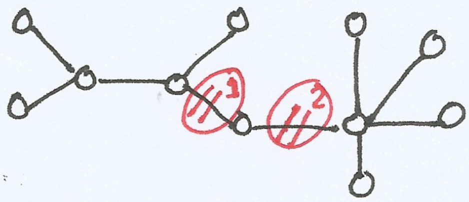
- Therefore the assumption that $\underline{T}$ was MST would have been violated if we could find such a T_1'

- So we have this nice property of optimal substructure:
  - I have subproblems that exhibit optimal solutions
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems [Cormen 2001]

- Question: What about overlapping subproblems for this type of problem?

• Yes, we will have overlapping subproblems

• Suppose that:



Performing the cuts in the following order:

• //1
• //2

Will produce the same MSTs when we perform the cuts in the following order:

• //2
• //1

• The same subproblem will repeat themselfes!!!

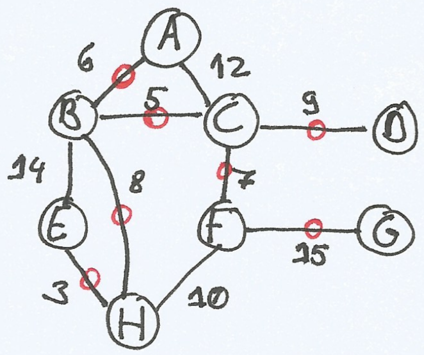
• This means we can use Dynamic Programming to remember (memoize) solutions that we have found

- Although we could use DP it turns out that MSTs exhibit a even more powerful property:

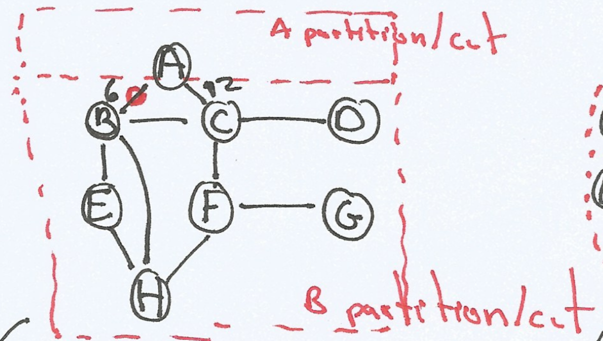
> **Greedy choice property:**
>
> A locally optimal choice is globally optimal

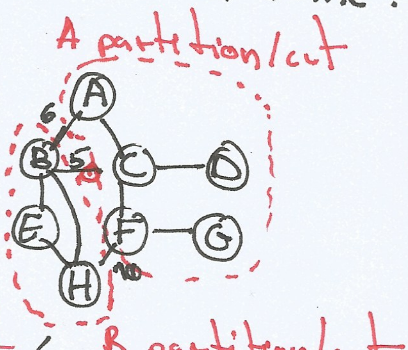
- It turns out that when we have the greedy choice property we can do better than dynamic programming

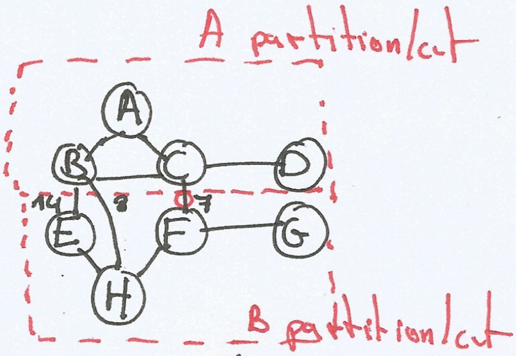
- **Theorem:**
  - Let T be the MST of $G = (V, E)$ and let $\overbrace{A \subseteq V}^{A \text{ is a subset of } V}$.
  - Suppose $(u, v) \in E$ is the least-weight edge connecting A to $V - A$.
  - Then $(u, v) \in T$ (the MST)
  - Example:



- The edges marked with a circle (o) are the ones that form the MST
- We can consider a multitude of scenarios lets look at some:







Then edge (A,B) belongs to the MST

Then edge (B,C) belongs to the MST

The edge (C,F) belongs to the MST

- I.e. picking that thing that is locally good for the subset A is also globally good (optimizes the overall function)

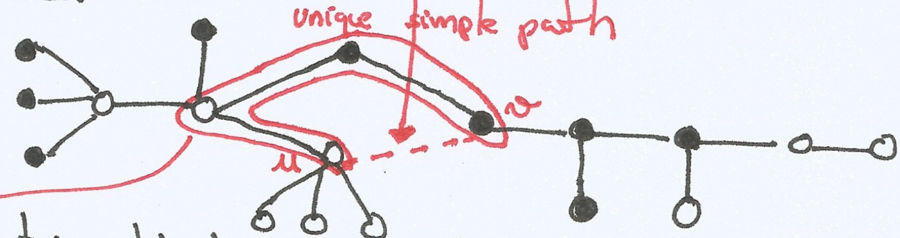
- **Proof:**

  - Suppose that $(u,v) \in E$ that is the least-weight edge connecting $\underline{A}$ to $\underline{V-A}$ is not in the $\underline{MST}$, i.e:

$$\underbrace{(u,v) \notin T}$$

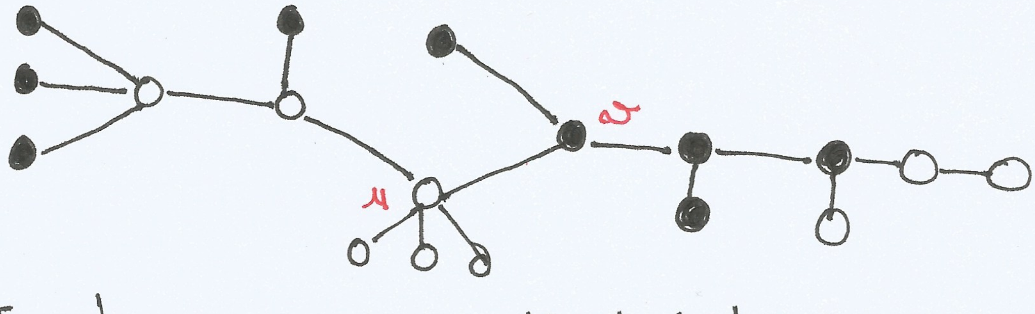
  - Example:

MST T:



Notation:
O ∈ A
● ∈ V − A

  - Notice that $u \in A$ and $v \in V-A$
  - We want to prove that (u,v) should be on the MST T
  - In a tree between any two vertices there is a unique simple path, i.e. it does not go back and forth and repeat edges or vertices
  - Consider unique simple path from $\underline{u}$ to $\underline{v}$ in $\underline{T}$. Swap (u,v) with first edge on this path that connects a vertex in $\underline{A}$ to a vertex in $\underline{V-A}$, i.e.:



  - The edge (u,v) is now the lightest edge connecting $\underline{A}$ to $\underline{V-A}$
  - Accordingly by performing the swap we obtained a lower weight MST, contradicting the assumption that $\underline{T}$ was

Prim's Algorithm

Idea: – Maintain $V-A$ as a priority Queue Q

– key each vertex in Q with weight of least-weight edge connecting it to a vertex in A

Q represent set $V-A$

Analysis:

$Q \leftarrow V$ (i.e. A starts out empty)

$key[v] \leftarrow \infty \quad \forall v \in V$

$key[s] \leftarrow 0$ for arbitrary s in V

$\Theta(|V|)$

while $Q \neq \emptyset$ .... $|V|$

do $u \leftarrow$ Extract-Min(Q) → $\log|V|$ using a Min-Heap

By the handshaking lemma we know that this is $2|E|$

for each $v \in Adj[u]$ .... $deg(u)$

do if $\underbrace{v \in Q}_{\Theta(1)\text{ time}}$ and $\underbrace{w(u,v) < key[v]}_{\Theta(1)\text{ time}}$

Keep a bit for each vertex that tells whether or not it is in Q. Update ~~vertex~~ bit when vertex is removed from Q

then $key[v] \leftarrow w(u,v)$

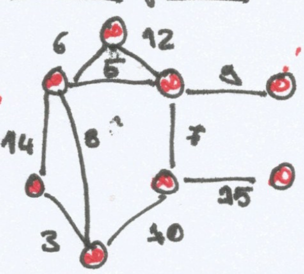
pi...... $\pi[v] \leftarrow u$

We need to update the key value in the priority Queue = Decrease key(newkey) $O(\lg n)$

At end, $\{(v, \pi[v])\}$ forms the MST

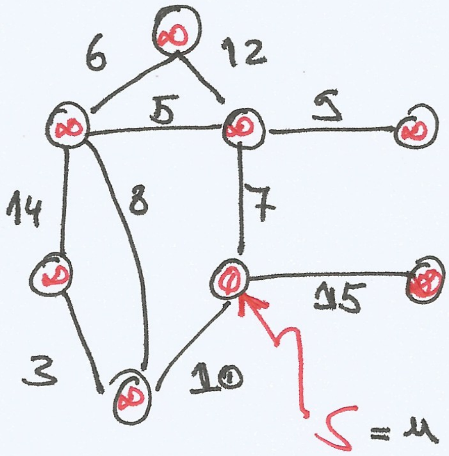
Example:

1st step



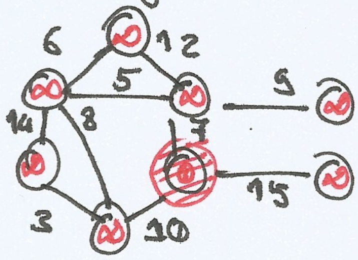
Everything starts out with ∞ value

2nd step:



3rd step:

- Extract-min implies that u will now join A



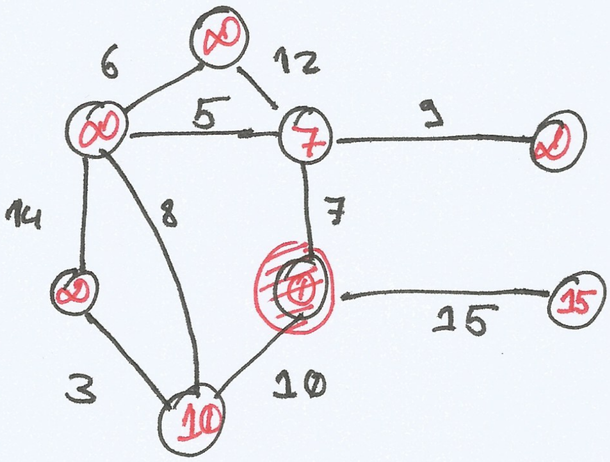
Notation:

$\in A$

$O \in V - A$

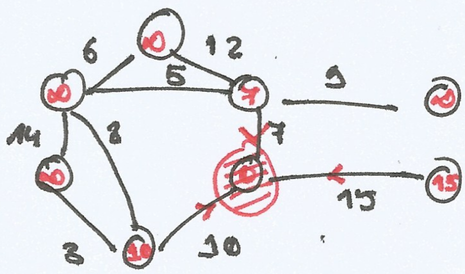
4th step:

- For each vertex in adj[u] we will check if it is still in Q (that is V − A) and $u(u,v) < key[v]$ then we are going to replace by the edge value $u(u,v)$



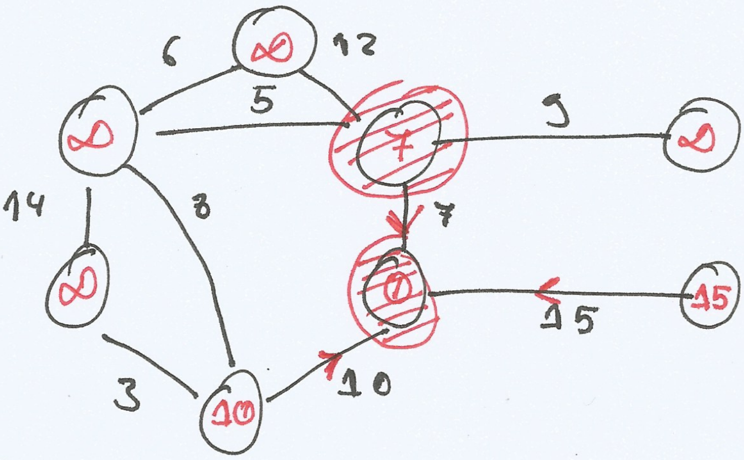
5th step:

- We need to setup the pointers back to u:



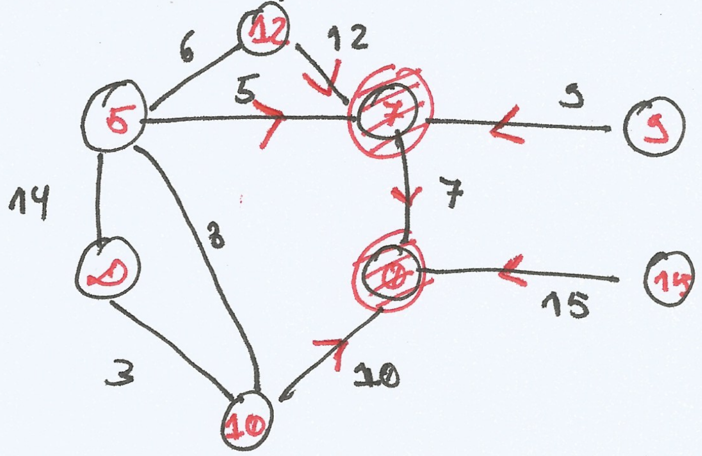
6th step:

Extract-Min

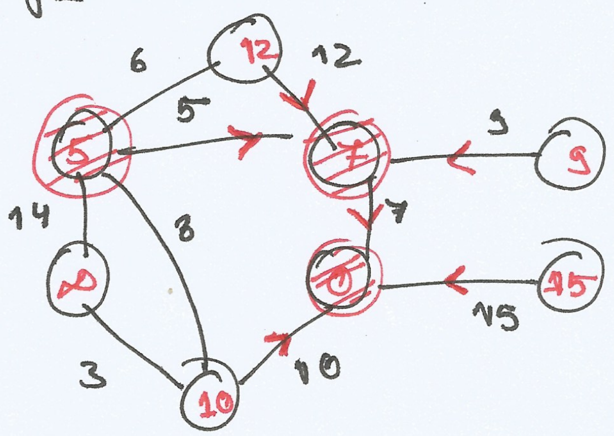


7th step:

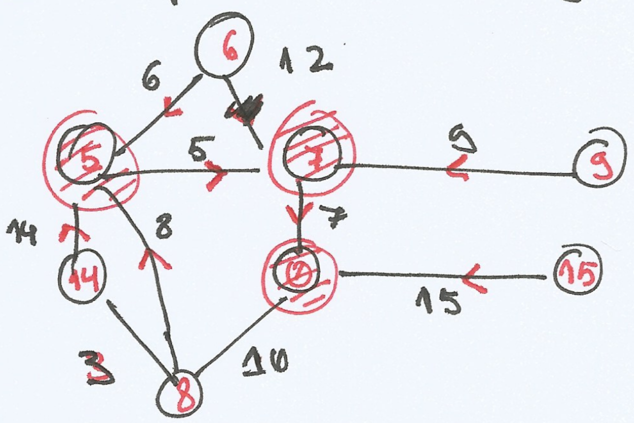
Now we update the neighbours and the parent pointers



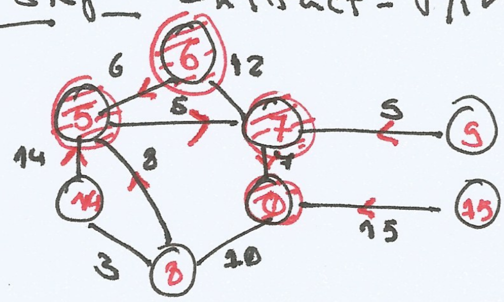
8th step: Extract-Min



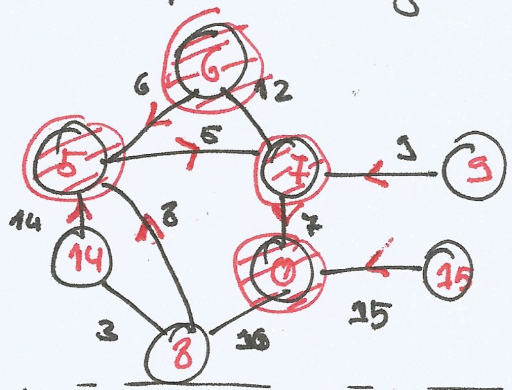
9th step: Update neighbours and parent pointers



10th step: Extract-Min

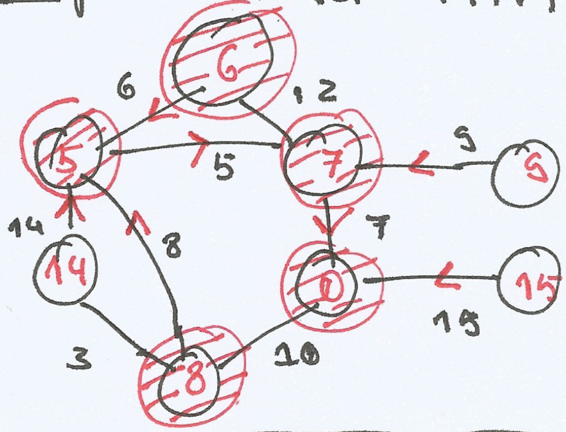


11th step: Update neighbours and parent pointers

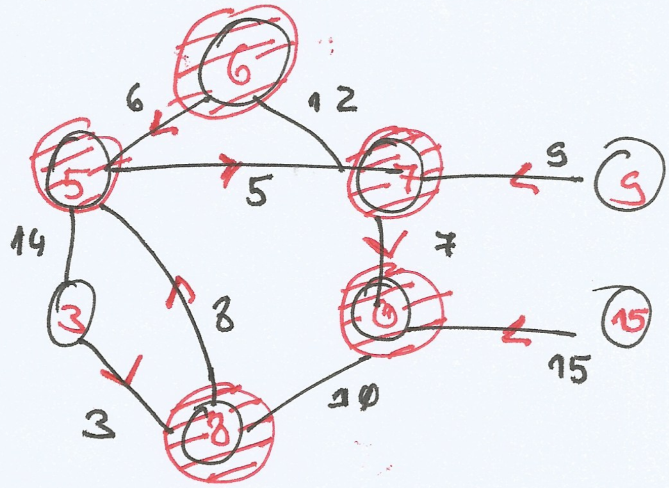


There is nothing to update since all the neighbours $\notin Q$ (i.e. $\in A$)

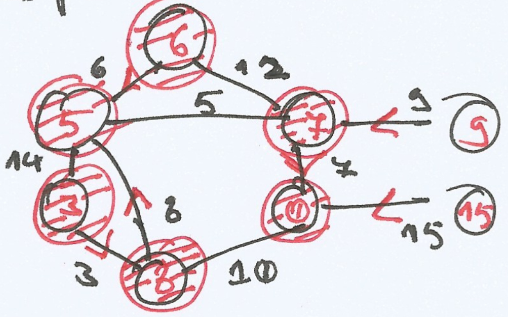
12th step: Extract-Min



13th step: Update neighbours and parent pointers

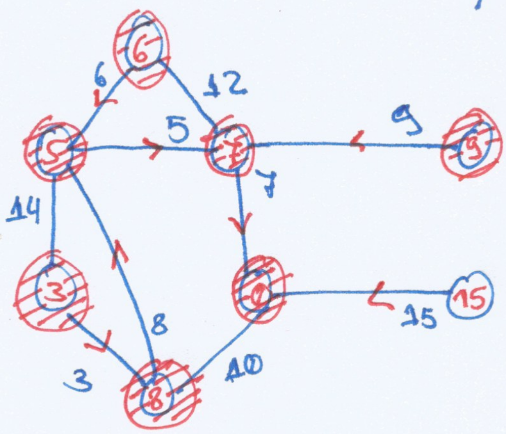


14th step: Extract-Min, update neighbours and parent pointers



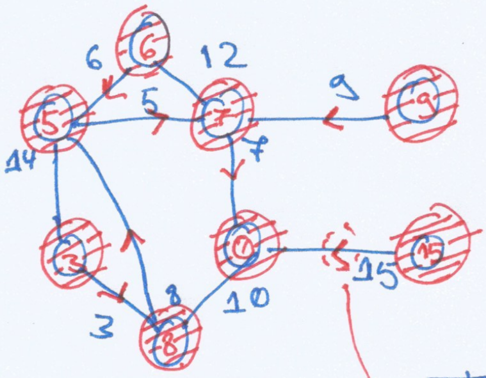
Nothing to be done

15th step: Extract-Min; update-neighbours and parent pointers



Nothing to be done

16th step: Extract-Min; update-neighbours and parent pointers



Nothing to be done and the algorithm terminates

The edges belonging to the MST are indicated by a < symbol

Question: What is the total running time?

Using the handshaking lemma

$$|V| + |V|\Big(\underbrace{\log |V|}_{\text{Also using a MinHeap}} + \overbrace{2|E|}^{} \log |V| \Big) \Rightarrow O\big(|V| + |V|\cdot \log |V| + |V|\cdot|E| \log |V|\big)$$

$= O\big(|V|\cdot \log |V| + |V||E|\cdot \log |V|\big)$ since the term $|V|$ is small when $n \to \infty$

We can make this thighter if we notice that the inner for loop in reality processes all the vertexes and not the edges. Then we obtain

$$|V| + |V|\big(\log |V| + |V|\cdot \log |V|\big) \Rightarrow O\big(|V|\cdot \log |V| + |V|^2 \cdot \log |V|\big)$$

But for dense graphs $|V|^2 \simeq |E|$, which allows us to obtain

$$O\big(|V|\cdot \log |V| + |E|\cdot \log |V|\big)$$

However, it is not uncommon for $V \ll E$ which allows to obtain:

$$O\big(|E|\cdot \log |V|\big)$$

- In reality through amortized analysis it can be shown that the overall time complexity is:

$$Time = \Theta(V \cdot T_{Extract\text{-}Min} + E \cdot T_{Decrease\text{-}key})$$

- I.e. it depends on how $\underline{Q}$ is implemented:

Q	$T_{Extract\text{-}Min}$	$T_{Decrease\text{-}key}$	Total
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O(\lvert E\rvert \log \lvert V\rvert)$
Fib Heap	$O(\log \lvert V\rvert)$ amortized	$O(1)$ amortized	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$