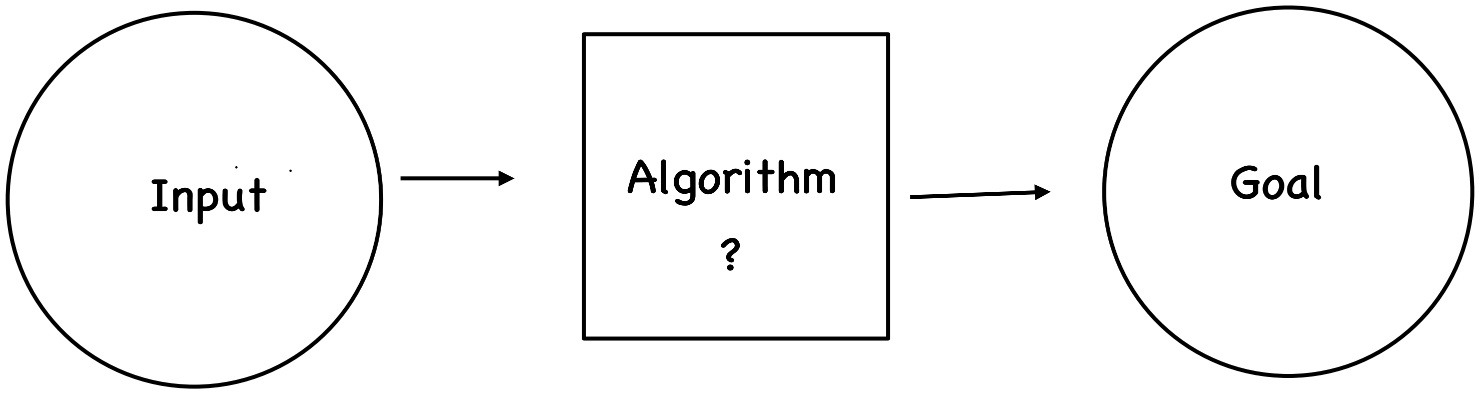
Lecture 2

Greedy algorithms



Given a problem, how do we design an algorithm that solves the problem? There are several strategies:

1. Try to modify an existing algorithm.
2. Construct an algorithm belonging to a special suitable class of algorithms with a given design pattern.
3. Develop an entirely new algorithm.

If we use strategy 1 we can, for instance, use one of these algorithms:

Graph algorithms
Flow algorithms
Linear Programming (Simplex method)

We will describe three classes of algorithms that can be used in case 2:

Greedy algorithms

Divide & Conquer algorithms

Dynamic Programming algorithms

But first, an example of how we can modify an existing algorithm. Remember DFS:

```
Set R = ∅
For all v ∈ V
    Set vis(v) = 0
End for
DFS(s)
```

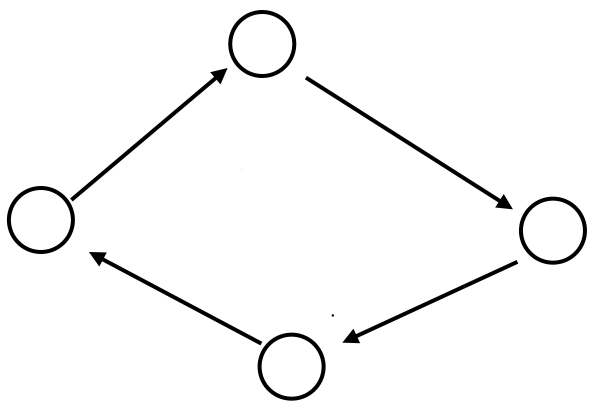
```
DFS(u):

Set vis(u) = 1
Add u to R
For each v such that v is adjacent to u
    If vis(v) = 0
        DFS(v)
    End if
End for
```

The algorithm can be used both for directed and undirected graphs.

DIRECTED CYCLE

Input: A directed graph G

Goal: Does G contain a directed cycle?



Modified DFS:

```
For all v ∈ V
    Set vis(v) = 0
End for
While there is v such that vis(v) = 0
    DFS_Mod(v)
Return "No "
```

```
DFS_Mod(u):

Set vis(u) = 0,5
For each v such that v is adjacent to u
    If vis(v) = 0,5
        Return "Yes"
    Else if vis(v) = 0
        DFS_Mod(v)
    End if
End for
Set vis(u) = 1
```

We can prove that this algorithm stops and returns Yes or No correctly.

We now describe greedy algorithms:

Let us say that I is an instance of a problem P. Let us assume that a solution to the problem can be represented as a sequence of choices C[1], C[2], ..., C[k]. So we want to find a good set of choices.

Let us also assume that when the first choice is made, the remaining choices form a new instance I' of the same problem.

A greedy algorithm is an algorithm that makes the first choice C[1] following some very simple strategy and then use this method recursively on I'. With simple we usually mean that we make a choice that seems good at this moment without looking ahead into the consequences of the choice.

A special case is when we have an optimization problem where we have some functional F depending on the choices C[1], ... and we want to maximize F. A greedy choice is then to choose C[1] so that the local increase in F is maximal.

Greedy algorithms should always have polynomial time complexity. Often they have linear time complexity.

Ex:
We have the numbers 10, 5 and 1. We are given the integer N. We want to write N as a sum of of the numbers 10, 5, 1. (We can use a number more than one time.) That is, we want to find numbers a,b,c such that N = a 10 + b 5 + c. Furthermore, we want to use as few terms as possible. That is, a + b + c should be as small as possible.

The solution is obvious: As long as N is greater than 9 we subtract 10 to get a new number N and repeat. When N is smaller than 10 we subtract 5 if possible. Then we subtract 1 until we reach 0. So, for instance, N = 37 gives a = 3, b = 1, c = 2. Obviously, we can not do better than this. This is a greedy algorithm.

A greedy algorithm can fail for two reasons:

1. It can fail to give us an optimal solution

Ex: We take the same problem as before, but instead of 10, 5, 1 we use the numbers 6, 5, 1. If we have N = 10, the greedy algorithm gives us 10 = 6 + 1 + 1 + 1+ 1. But the best solution is 10 = 5 + 5.

2. It can fail to give us a solution

Ex: The same problem but with the numbers 6, 5, 2. If we take N = 7, the greedy algorithm subtracts 6 from 7 and leaves us with 1. Then the algorithm fails to reach the sum 7. The correct solution is 7 = 5 + 2.

But when do greedy algorithms work? We study some examples.

Ex:
We want to drive along a road. We represent the road as a coordinate axis. We start at x = 0 and want to go to a city x[n]. Along the road there are other cities x[1], x[2], ...x[n-1]. A full gas tank contains gas for A kilometers. We can fill the tank in the cities but nowhere else. We want to reach x[n] and tank as few times as possible. How do we do that?

We might think that we should use some complicated strategy but that is not so. In fact, a greedy algorithm works:

```
Set L = Ø
Set i = 0
Set T = A
While TRUE do
    While x[i+1]- x[i] =<  T and i  <  n  do
        Set T = T - (x[i+1] - x[i])
        Set i = i + 1
    End while
    If i = n then
        Halt
    If x[i+1] - x[i] <  A then
        Return "Impossible"
    Set T = A
    Put i at the end of L
End while
```

The time-complexity is O(n).

If we are at x[i] and have enough gas left to reach x[i+1] we do not fill gas. Otherwise, we get a full tank at x[i]. If it is at all possible to get to x[n], this algorithm will take us there and fill gas as few times as possible.

A usual way show that a greedy algorithm is correct is to show that the first choice C[1] cannot be wrong (and inductively no other choices either). Let us assume that S* is a solution other than our and S* does not contain C[1]. We might then try to rearrange S* very little and obtain a new solution S' which does contain C[1]. If this can be done, the greedy algorithm is correct. ,

We will look at some more examples:

Activity planning

Let us assume that we have n activities $a_1, a_2, ..., a_n$ with corresponding time intervals $[s_i, f_i)$. No intervals are allowed to overlap each other. (The intervals are half-open. Observe that $[2, 4)$ och $[4, 5)$ do not overlap.) How do we choose a maximal number of activities that do not overlap each other?

Greedy algorithm for activity planning

It turns out that we shall choose activities after end times. This algorithm chooses a set A of activities. Sort the activities such that $f_1 \leq f_2 \leq ... \leq f_n$.

```
(1)  $A \leftarrow \{a_1\}$
(2)  $i \leftarrow 1$
(3)  **for** $j \leftarrow 2$ **to** n
(4)     **if** $s_j \geq f_i$
(5)        $A \leftarrow A \cup \{a_j\}$
(6)        $i \leftarrow j$
(7)  **return** $A$
```

The time-complexity is O(n).

Jobs with deadlines

Let us assume that we have n jobs which must be done by one person. It takes time t_i to do job i. We also know that job i must be finished latest at time d_i. We want to plan times for doing the jobs such that:

- $f(i) = s(i) + t_i$ for all i.
- No intervals $[s(i), f(i)]$, $[s(j), f(j)]$ overlap each other.
- $f(i) \leq d_i$ for all i.

A first problem is to decide if this is possible and how the planning then looks.

If the planning is impossible to make, this must be because $f(i) > d_i$ for some i. We can try to minimize the failure". There are several ways of measuring the failure. A natural idea is the following:

Set L $= \max\limits_i f_i - d_i$.

Then try to get L as small as possible.

The algorithm is really simple. Sort the job so that $d_1 \leq d_2 \leq \cdots \leq d_n$. We assume that $d_i > 0$ for all i and that the first job starts at 0.

```
(1)  $s(1) \leftarrow 0$, $f(1) \leftarrow t_1$
(2)  for $i \leftarrow 2$ to n
(3)      $s(i) \leftarrow f(i-1)$, $f(i) \leftarrow s(i)+t_i$
(4)  return $s, f$
```

The Minimal Spanning Tree Problem

If G is a connected graph,then a spanning tree is a tree that contains all nodes in G.

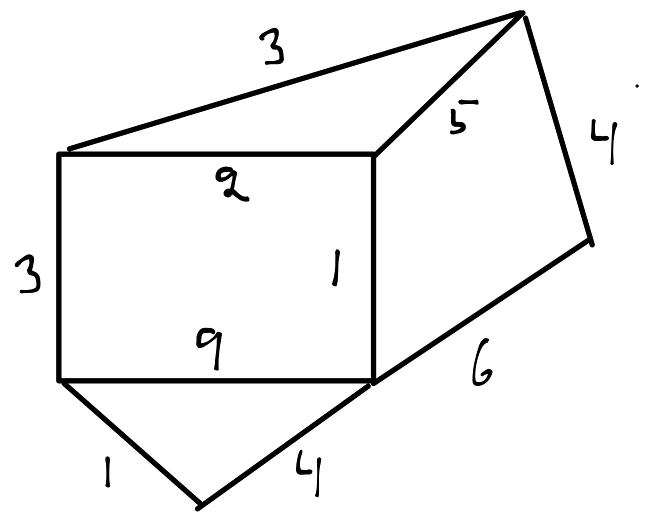
Obs: If $|V| = n$ and $T \subseteq G$ is a tree then

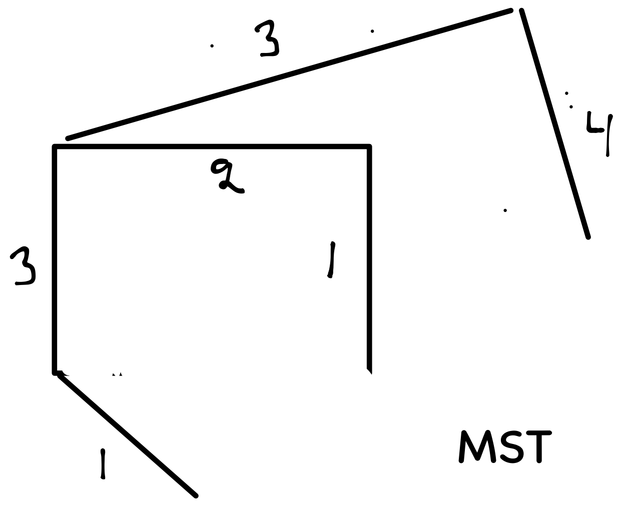
T is spanning <=> $|E| = n - 1$

Let us take a weighted graph.

A minimal spanning tree (MST) is a spanning tree such that

$$W = \sum_{e \in E(T)} w(e)$$





MST

The MST problem:

Input: W weighted connected graph G
Goal: A MST in G

There are two famous greedy algorithms for solving this problem: Kruskal's algorithm and Prim's algorithm.

Kruskal's algorithm

```
Sort the edges such that $w(e_1) \le w(e_2) \le \dots$
Set $A = \emptyset$
For each $e_i$ in the sorted order
    If $A \cup \{e_i\}$ does not contain any cycle
        Set $A = A \cup \{e_i\}$
    End if
End for
Return A
```

A first form

How do we decide the complexity? How do you know if a set of edges contains a cycle or not? We have to describe the algorithm more in details.

Data structures for identifying cycles:

We represent disjoint sets of nodes with a balanced tree. One of the nodes is the root of the tree. This node will be the "name" of the set. In practice, the tree is implemented by pointers. Each node has a pointer pointing to its father.

We have the following operations:

MakeSet(v) creates the set {v}
This is just a tree with one mode.
Complexity: O(1)

FindSet(v) finds the set containing v
It start with v and finds a path (maximum length log | V |) to the name of the set.
Complexity : O(log |V|)

Make Union(u,v) makes the union of the sets containing u and v
Here we must merge two trees. Basically, the shorter tree is added to the root, giving a new branch.
Complexity : O(log |V|)

An implementation could look like this:

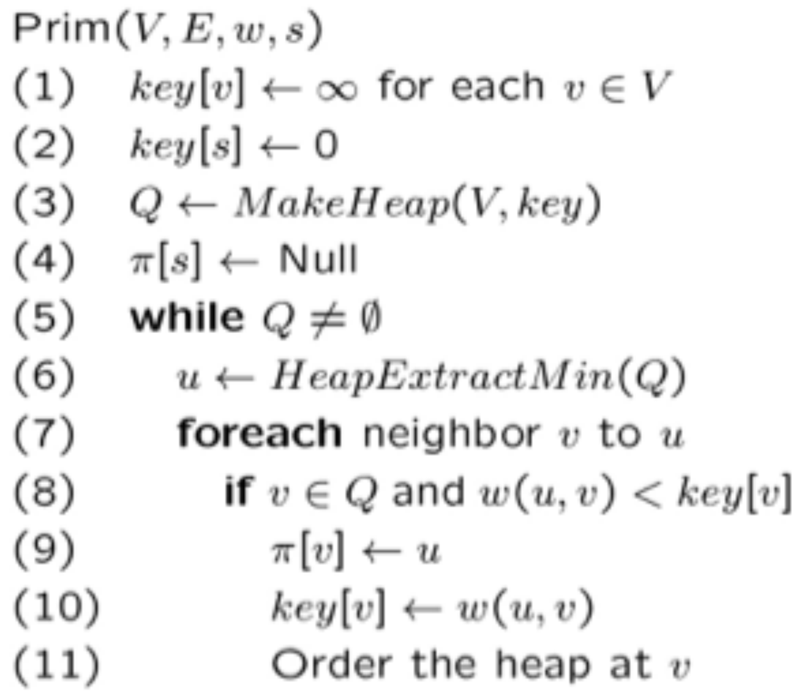
```
MakeSet(v) =
    p(x) = x
    rank(x) = 0

FindSet(x) =
    while x = p(x)
        x = p(x)
    Return x

MakeUnion(x,y) =
    rx = FindSet(x)
    ry = FindSet(y)
    if rx = ry
    Return
    if rank(rx) > rank(ry)
        p(ry) = rx
    else
        p(rx) = ry
        if rank(rx) = rank(ry)
            rank(ry) = rank(ry) + 1
```

```
Kruskal($V, E, w$)
(1)  $A \leftarrow \emptyset$
(2)  **foreach** $v \in V$
(3)     MakeSet($v$)
(4)  Sort $E$ in increasing weight order
(5)  **foreach** $(u, v) \in E$ (in the sorted order)
(6)     **if** FindSet($u$) $\neq$ FindSet($v$)
(7)        $A \leftarrow A \cup \{(u, v)\}$
(8)        MakeUnion($u, v$)
(9)  **return** $A$
```

Complexity: $O(|E| \log |E|)$ (due to the sorting); FindSet and MakeUnion takes $O(|E| \; log|V|)$ tid.

Another similar algorithm is Prim's algorithm

Here we use a heap. A heap is essentially a balanced binary tree with numbers in at each node. Each node has a number no greater than those of its children's. The root will contain the smallest number in the heap. The heap can be adjusted effectively if one of the numbers change.



```
Prim($V, E, w, s$)
$key[v] \leftarrow \infty$ for each $v \in V$
$key[s] \leftarrow 0$
$Q \leftarrow MakeHeap(V, key)$
$\pi[s] \leftarrow$ Null
while $Q \neq \emptyset$
    $u \leftarrow HeapExtractMin(Q)$
    foreach neighbor $v$ to $u$
        if $v \in Q$ and $w(u, v) < key[v]$
            $\pi[v] \leftarrow u$
            $key[v] \leftarrow w(u, v)$
            Order the heap at $v$
```

It can be showed that the complexity is O(|E| log |V|)

Proof of correctness: For simplicity we assume that all edges have different weights. We use the following lemma: Assume that all edges have different weight. A cut S in G is a partitioning V = S ∪ (V-S).

Theorem.
Let S, V-S be a cut (none of S or V-S is empty). Let e = (u,v) be an edge from S till V-S and such that e:s weight is minimal. Then e must be in every MST in G.

Proof: Let T be a MST that does not contain (u,v). There is a path from u to v in T. The path contains some edge (x, y) going from S to V-S. Since (u,v) has least weight of all such edges we see that T + {(u, v)}-{x, y)} is a MST with lower weight, which is impossible.

Kruskal: Let (u,v) be an edge chosen at any stage in Kruskal. Let S be all nodes that can be reached from u be paths using edges already chosen. Then u ∈ S and v ∉ S. We can see that (u,v) must be an edge of minimal weight cross the cut. The theorem says that this edge must be in all MST:s. Then the choice of (u,v) can not be wrong.
Prim: Let S be the nodes chosen at a certain stage. Let (u,v) be the edge chosen in the next step.Then the theorem says that this edge must be in all MST:s. Then the choice of (u,v) can not be wrong.

The proof can be modified to cover the case when edges are allowed to have equal weights.