Greedy algorithms

In this lecture we will describe a general template for finding algorithms. It works in a surprisingly large number of cases. It's the method of greedy algorithms.

We will study a special type of problems. We want to make choices. Let's say that we make choices c[1], c[2], c[3], ... At each step in the algorithm we have a set of possible choices. When the algorithm ends we want to have a selection c[1], c[2], c[3], ...c[k] that in some sense is correct. We assume that there, to each selection, is associated a cost A. Let's assume that our goal is to find a correct selection with as small cost as possible.

Furthermore, we assume that when the choice c[1] is made the remaining situation is a problem of the same type. (This notion seems hard define in a precise way.) Then there is a chance that a so called greedy algorithm will work.

A greedy algorithm is a an algorithm which make the choices following a very simple (greedy) strategy. What this means depends on the situation. Usually there are two sorts of greedy strategies:

1. We can make the choice c[1] so that the cost (locally) increases as little as possible.

2. We can make the choice c[1] so that the remaining problem is as "good" as possible.

(The first case is what we in the strictest sense means by a greedy algorithm. But lots of interesting problems are covered by the second, more vague case.)

The greedy algorithm runs like this: Assume that we have made choices c[1], c[2], c[3], ..., c[m]. If this selection is correct we stop. Otherwise, make the next choice following your greedy strategy.

The general idea with a greedy algorithm is that you don't have to spend long time on making your choices. You don't have to look ahead and consider the consequences. Greedy algorithm usually have low 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.

When greedy algorithms fail

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 correct 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:

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.

The time-complexity is O(n).

We will look at some more examples:

```
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
```

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 ← {a_1}
(2)  i ← 1
(3)  for j ← 2 to n
(4)     if s_j ≥ f_i
(5)        A ← A ∪ {a_j}
(6)        i ← 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 $= \max_i f_i - d_i$.

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.

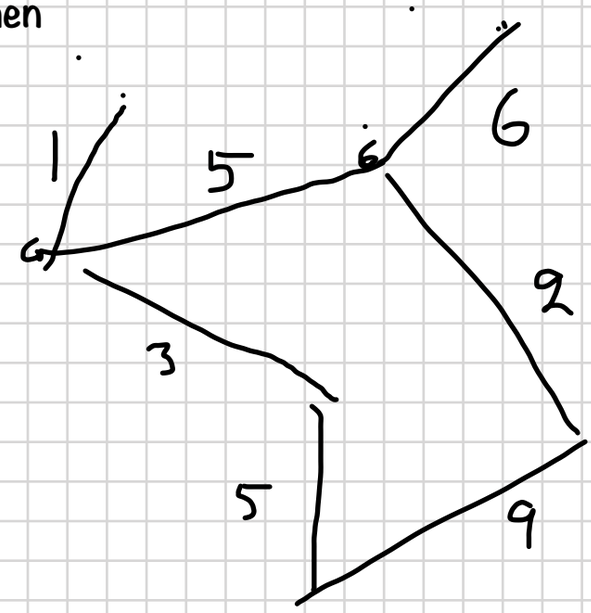
(1) $s(1) \leftarrow 0,\ f(1) \leftarrow t_1$
(2) **for** $i \leftarrow 2$ **to** n
(3) $\quad s(i) \leftarrow f(i-1),\ f(i) \leftarrow s(i)+t_i$
(4) **return** s, f

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 $\Leftrightarrow |E| = n - 1$



A graph with node weights

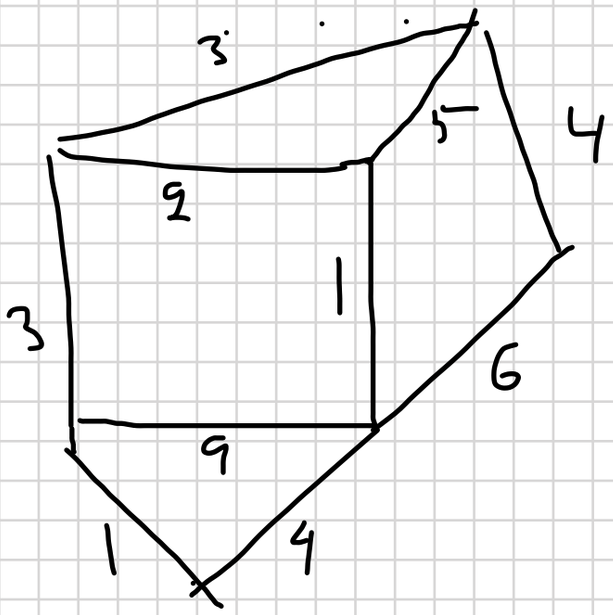
A minimal spanning tree (MST) is a spanning tree such that

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

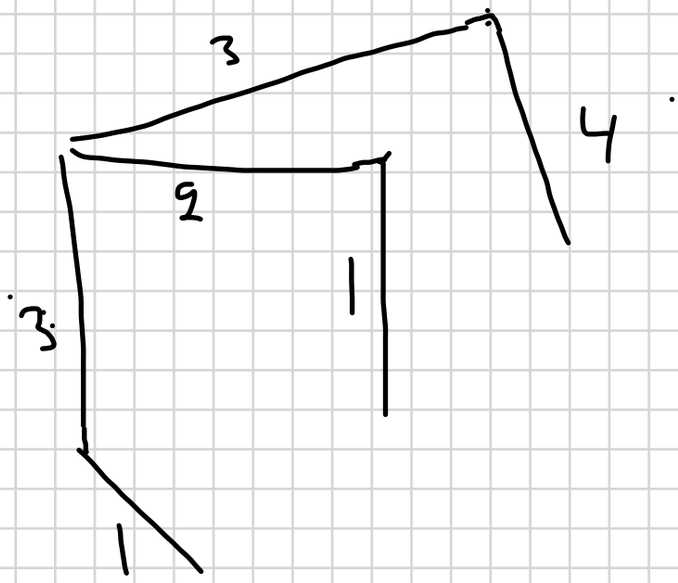
is minimal.

The MST problem:

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



MST



Kruskal's algorithm

Sort the edges such that $w(e_1) \leq w(e_2) \leq \ldots$
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

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:

MakeSet(v) creates the set {v}
    Complexity: O(1)

FindSet(v) finds the set containing v
    Complexity : O(log |V|)

Make Union(u,v) makes the union of the sets containing u and v
    Complexity : O(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

```
Prim(V, E, w, s)
(1)   key[v] ← ∞ for each v ∈ V
(2)   key[s] ← 0
(3)   Q ← MakeHeap(V, key)
(4)   π[s] ← Null
(5)   while Q ≠ ∅
(6)     u ← HeapExtractMin(Q)
(7)     foreach neighbor v to u
(8)       if v ∈ Q and w(u, v) < key[v]
(9)         π[v] ← u
(10)        key[v] ← w(u, v)
(11)        Order the heap at v
```

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