Lecture 13 - General problem-solving methods

- **Greedy**
- **Dynamic programming**

Greedy Method

- a strategy to solve optimization problems
- applicable where the global optima may be found by successive selections of local optima
- allows solving problems without returns to the previous decisions.
- useful in solving many practical problems that require the selection of a set of elements that satisfies certain conditions (properties) and realizes an optimum

Sample problems

The knapsack problem

A set of objects is given, characterized by usefulness and weight, and a knapsack able to support a total weight of W. We are required to place in the knapsack some of the objects, such that the total weight of the objects is not larger that the given value W, and the objects should be as useful as possible (the sum of the utility values is maximal).

The coins problem

Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) *units* (*an unlimited number of coins*)*. The problem is to establish a modality to pay the sum M using a minimum number of coins.*

General abstraction for a Greedy-like problem

Let us consider the given set C of candidates to the solution of a given problem P. We are required to provide a subset B (B $\subseteq$ C) to fulfill certain conditions (called internal conditions) and to maximize (minimize) a certain objective function.

- If a subset *X* fulfills the internal conditions we will say that the subset *X* is acceptable (possible).
- Some problems may have more acceptable solutions, and in such a case we are required to provide an as good a solution as we may get, possibly even the better one, i.e. the solution that realizes the maximum (minimum) of a certain objective function.

In order for a problem to be solvable using the Greedy method, it should satisfy the following property:

- if *B* is an acceptable solution and $X \subseteq B$ then *X* is as well an acceptable solution.

Greedy algorithm

The Greedy algorithm find the solution in an incremental way, by building acceptable solutions, extended continuously. At each step, the solution is extended with the best candidate from *C*-*B* at that given moment. For this reason, this method is named greedy.

The Greedy principle (strategy) is

- to successively incorporate elements that realize the local optimum
- no second thoughts are allowed on already made decisions with respect to the past choices.

Greedy algorithm

By assuming that Ø (the empty set) is an acceptable solution, we will construct the set *B* by initializing B with the empty set and successively adding elements from *C*.

The choice of an element from *C*, with the purpose of enriching the acceptable solution *B*, is realized with the purpose of achieving an optimum for that particular moment, and this, by itself, does not generally guarantee the global optimum.

If we have discovered a selection rule to help us reach the global optimum, then we may safely use the Greedy method.

there are situations in which the completeness requirements (obtaining the optimal solution) are abandoned in order to determine an "almost" optimal solution, but in a shorter time.

Greedy technique:

1. renounces to the *backtracking* mechanism.
2. offers a single solution (unlike *backtracking*, that provides all the possible solutions of a problem).
3. provides a polynomial running time.

Greedy – sample python code

```
def greedy(c):
    """
       Greedy algorithm
       c - a list of candidates
       return a list (B) the solution found (if exists) using the greedy 
strategy, None if the algorithm
       selectMostPromissing - a function that return the most promising 
candidate
       acceptable - a function that returns True if a candidate solution can be 
extended to a solution
       solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Greedy algorithms essential elements

1. A ***candidate set***, from which a solution is created;
2. A ***selection function***, which chooses the best candidate to be added to the solution;
3. A ***feasibility function***, that is used to determine if a candidate can be used to contribute to a solution;
4. An ***objective function***, which assigns a value to a solution, or a partial solution, and;
5. A ***solution function***, which will indicate when we have discovered a complete solution.

The coins problem

Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) *units* (*an unlimited number of coins*)*. The problem is to establish a modality to pay the sum M using a minimum number of coins.*

Solution:

Candidate Set:

The list of coins - COINS = {1, 5, 25, 50}

Candidate Solution:

A list of selected coins - $X=(X_{0,}X_{1,.},X_k)$ **where** $X_i \in COINS$ **– used coin**

Selection Function:

candidate solution: $X=(X_{0,}X_{1,.},X_k)$

choose the biggest coin less than the remaining sum to pay

Acceptable (*feasibility function)*:

The sum payed with the current coins are not exceeding M

Candidate solution: $X=(X_{0,}X_{1,.},X_k)$ $S=\sum_{(i=0)}^{k} X_i \le M$

Solution function:

The sum payed with the coins in X is equal with M

Candidate solution: $X=(X_{0,}X_{1,.},X_k)$ $S=\sum_{(i=0)}^{k} X_i = M$

The coins problem – python code

```
#Let us consider that we have a sum M of money and coins of 1, 5, 25 units (an unlimited number of coins).
#The problem is to establish a modality to pay the sum M using a minimum number of coins.
```

```
def selectMostPromissing(c):
    """
      select the largest coin from the remaining
      c - candidate coins
      return a coin
    """
    return max(c)
```

```
def acceptable(b):
    """
   verify if a candidate solution is valid
   basically verify if we are not over the sum M
    """
    sum = _computeSum(b)
    return sum<=SUM
```

```
def solution(b):
    """
   verify if a candidate solution is an actual solution
   basically verify if the coins conduct to the sum M
   b - candidate solution
    """
    sum = _computeSum(b)
    return sum==SUM

def _computeSum(b):
    """
   compute the payed amount with the current candidate
    return int, the payment
   b - candidate solution
    """
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        #if this is in a candidate solution we need to
use at least 1 coin
        if nrCoins==0: nrCoins =1
        sum += nrCoins*coin
    return sum
```

```
def printSol(b):
 """
   Print the solution: NrCoinns1 * Coin1 +  NrCoinns2 *
Coin2 +...
 """
    solStr = ""
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        solStr+=str(nrCoins)+"*"+str(coin)
        sum += nrCoins*coin
        if SUM-sum>0:solStr+=" + "
    print solStr
```

Remarks

1. It is necessary that before applying a Greedy algorithm to **prove** that it will provide the optimal solution. Often, the proof of applicability of Greedy technique can be a non-trivial process.
2. Greedy technique leads to a polynomial running time. Usually, if the cardinality of the set C of candidates is n, Greedy algorithms have $O(n^2)$ time complexity.
3. There are a lot of problems that can be solved using Greedy technique: determining the minimum spanning tree in a graph (Kruskal's algorithm), determining the shortest path between two nodes in an undirected or directed graph (Dijkstra's algorithm and Bellman-Kalaba's algorithm, respectively).
4. There are problems for which Greedy algorithms do not provide the optimal solution. In some cases, is preferable to obtain in a reasonable (polynomial) time a solution very close to the optimal solution, instead of obtaining the optimal solution in an exponential time. These algorithms are called *heuristics algorithms*.

Dynamic programming method

Applicable in solving optimality problems where

- the solution is the result of a sequence of decisions, $d_1, d_2, \ldots, d_n$,
- the *principle of optimality* holds.
- usually leads to a polynomial running time
- always provides the optimal solution (unlike Greedy).
- like the data division method solves problems by combining the sub solutions of their sub problems, but, unlike it, calculates only once a sub solution, by storing the intermediate results.

We consider the states $s_0, s_1, \ldots, s_{n-1}, s_n$, where s_0 is the initial state, and s_n is the final state, obtained by successively applying the sequence of decisions $d_1, d_2, \ldots, d_n$ (using the decision d_i we pass from state s_{i-1} to state s_i, for i=1,n):

$$\begin{tikzcd} s_0 \arrow[r, "d_1"] & s_1 \arrow[r, "d_2"] & s_2 \arrow[r] & \ldots \arrow[r, "\ldots"'] & s_{n-1} \arrow[r, "d_n"] & s_n \end{tikzcd}$$

Dynamic programming method

Dynamic programming method makes use of three main issues :

- the principle of optimality;
- overlapping sub problems;
- *memoization***.**

The principle of optimality

- the *general optimum* implies *partials optima*
- in an optimal sequence of decisions, each decision is optimal.
- is not always satisfied, especially in cases when sub-sequences are not independent and optimization of one of them is in conflict with the optimization of other.

The principle of optimality

If $d_1, d_2, ..., d_n$ is a sequence of decisions that optimally leads a system from the state s_0 to the state s_n, then one of the following conditions has to be satisfied:

1). $d_k, d_{k+1}, ..., d_n$ is an optimal sequence of decisions that optimally leads the system from the state s_{k-1} to the state s_n, $\forall k, 1 \leq k \leq n$. **(*forward* variant)**

2). $d_1, d_2, ..., d_k$ is a sequence of decisions that optimally leads the system from the state s_0 to the state s_k, $\forall k, 1 \leq k \leq n$. **(*backward* variant)**

3). $d_{k+1}, d_{k+2}, ..., d_n$ and $d_1, d_2, ..., d_k$ are sequences of decisions that optimally lead the system from the state s_k to the state s_n and, respectively, from the state s_0 to the state s_k, $\forall k, 1 \leq k \leq n$. **(*mixed* variant)**

Overlapping Sub-problems

A problem is said to have overlapping sub-problems if it can be broken down into sub-problems which are reused multiple times

Memorization

store the solutions to the sub-problems for later reuse

Applying Dynamic programming

- The principle of optimality (in one of its form: forward, backward or mixed) is proved.
- The structure of the optimal solution is defined.
- Based on the principle of optimality, the value of the optimal solution is recursively defined. This means that recurrent relations, indicating the way to obtain the general optimum from partial optima, are defined.
- The value of the optimal solution is computed in a bottom-up manner, starting from the smallest cases for which the value of the solution is known.

Longest increasing subsist

Let us consider the list $a_1, a_2, ..., a_n$. *Determine the longest increasing subsist* $a_{i_1}, a_{i_2}, ..., a_{i_s}$ *of the list a.*

Solution:

- *The principle of optimality*
  - The principle of optimality is verified in its *forward* variant
- *The structure of the optimal solution*
  - We will construct two sequences: $l = < l_1 . l_2 , ... l_n >$ and $p = < p_1 . p_2 , ... p_n >$. The significance is the following:
    - l_k is the length of the longest increasing sub sequence that starts with element a_k.
    - p_k is the index of the element from sequence a that immediately follows a_k in the longest increasing sub-sequence that starts with element a_k.
- *The recursive definition for the value of the optimal solution*

– $l_n = 1, p_n = 0$

– $l_k = \max\{1 + l_i \mid a_i \geq a_k, k + 1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ...1$

– $p_k = \arg\max\{1 + l_i \mid a_i \geq a_k, k + 1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ..., 1$

Longest increasing subsist – python code

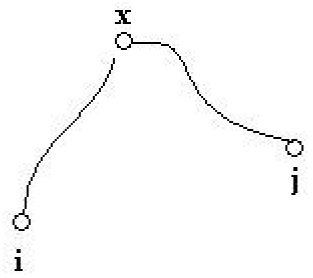
```
def longestSublist(a):
    """
      Determines the longest increasing sub-list
      a -  a list of element
      return sublist of x, the longest increasing sublist

    """
    #initialise l and p
    l = [0]*len(a)
    p = [0]*len(a)
    l[lg-1] = 1
    p[lg-1]=-1
    for k in range(lg-2, -1, -1):
        print p, l
        p[k] = -1
        l[k]=1
        for i in range(k+1, lg):
            if a[i]>=a[k] and l[k]<l[i]+1:
                l[k] = l[i]+1
                p[k] = i

    #identify the longest sublist
    #find the maximum length
    j = 0
    for i in range(0, lg):
        if l[i]>l[j]:
            j=i
    #collect the results using the position list
    rez = []
    while j!=-1:
        rez = rez+[a[j]]
        j = p[j]
    return rez
```

Dynamic programming vs. Greedy

- both techniques are applied in optimization problems
- Greedy is applicable to problems for which the *general optimum* is obtained from *partial* (*local*) *optima*
- DP is applicable to problems in which the *general optimum* implies *partial optima*.

Let us consider the problem to determine the optimal path between two vertices **i** and **j** of a graph. For this problem, we notice the following:

- *The principle of optimality* is verified: if the path from **i** to **j** is optimal and it passes through node **x**, then the paths from **i** to **x**, and from **x** to **j** are also optimal.



Dynamic programming vs. Greedy

- DP technique can be applied for solving this problem
- Greedy technique is not applicable, because: if the paths from **i** to **x**, and from **x** to **j** are optimal, there is no guarantee that the path from **i** to **j** that passes through **x** is also optimal.

Remarks

1. The fact that the general optimum implies the partial optimum, does not means that partial optima also implies the general optimum (the above example is relevant).
2. The fact that the general optimum implies partial optima is very useful, because we will search for the general optimum among the partial optima, which are stored at each moment. Anyway, the search is considerably reduced.