Algorithms and Data Structures for Biology
27 may 2019 — Assignment number 2

Daniele Traversa; ID:0000830457

1 The Problem

We want to deal with the following problem. Suppose you are the head of just-launched genomics research lab, and you need to decide which ones (between many) software packages to buy. After some analysis, you conclude that the market offers n software packages, each of them with a price of P_i Euros, and offering some functionalities: unfortunately, not all packages are equivalent! We can model the functionalities your lab needs as a set A = $\{a_1, ..., a_m\}$ where each a_i is distinct. As an example a1 could be some form of genome sequencing, while a_2 could be a 3D molecule rendering. Each software package thus offers a set of functionalities $S_i \subseteq A$. As an example, we could have that

$$P_3{=}156 \text{ ; } S_3{=}\{a_1, a_3, a_4\}$$

meaning that the third package costs 156 Euros, and offers the functionalities a_1,a_3 and a_4. Your goal, of course, is to decide which software packages you want to buy, so that all functionalities are somehow covered by at least one of the software packages you buy. Moreover, you want to do that minimizing the price.

2 Combinatorial problem

Given two sets `U`=$\{1.....n\}$ and an universe one $\{1.....m\}$. To each element $e \in$ `U` we associate an integer natural number P_e and a set S_e containing elements of the set $\{1.....m\}$ such that $S_e \subseteq \{1....m\}$. Find the set of elements G ⊆ U such that:

$$\bigcup_{S_e \in G} = \{1.....m\} \text{ and } \sum_{e \in G} P_e \text{ is minimal}$$

To set the combinatorial problem, different sources has been adopted [1]. The algorithm can be developed using two different strategies:

1. A `Branch And Bound` strategy can be applied. By means of that, the goal is to create in a incremental way, possible combinations of subsets of `U` and among them, we find the exact subset providing the optimal solution such that: $\bigcup_{S_e \in G} = \{1.....m\}$ and $\sum_{e \in G} P_e$ is minimal, where e stands for an element of `U` present in the subset G;

2. A `Greedy` strategy can be used. In this way, the algorithm provides us an approximate solution such that the subset `G` found by the algorithm is such to have $\bigcup_{S_e \in G} = \{1.....m\}$ but the $\sum_{e \in G} P_e$ may not minimal. In this way, the solution found is not the optimal one. Also in this case, e stands for an element of `U` present in the subset G.

3 The Algorithm-Branch And Bound

`REQUIRE`: An integer number `n` , an other integer number `m` generating the universe set $\{1.....m\}$, a set `S`$=\{S_1.....S_n\}$ and an other set `P`$=\{P_1......P_n\}$ both containing `n` elements.

`ENSURE`: Find the set of elements G $\subseteq$ U such that $\bigcup_{S_e \in G} = \{1.....m\}$ and $\sum_{e \in G} P_e$ is minimal, where U is the set $\{1.....n\}$.

One way to solve this problem is to construct the algorithm in a way that follows a `Branch And Bound` strategy, that is based on the exploration of a tree, where each internal node corresponds to all subsets of $\{1.....n\}$ until we reach the terminal leaves of the tree which represents the subsets having highest length possible . How does it work? Basically, the algorithm has to define the search space incrementally,so during the iteration process and, at the same time, it has to look for the optimal solution. To do that, it considers the set `U`$=\{1.....n\}$ that contains `n` elements. From this, it generates those subsets of the set `U` having size equal to 1. It starts with the first one and from this it explores all internal nodes of the tree until it reaches the final leave of the current branch of the tree. Of course the resulting subsets in each node have different lengths that go from 1 up to `n`. During the generation of the subsets it checks wherever the current node, called as *node* respects specific conditions which are:

$$\bigcup_{S_e \in node} = \{1......m\} \text{ and } \sum_{e \in node} P_e \text{ is minimal}$$

If the node that has generated has a $\sum_{e \in node} P_e$ that is already higher then the one of the node previously generated by the algorithm, actually there is no need to analyze nodes originating from the one that we have just created since all will have a $\sum_{e \in node} P_e$ that is even higher. So the point is that the algorithm skips the calculation of those branching nodes and proceeds in analyzing other nodes having the same size with respect to the one that we have just skipped. Otherwise, if the $\sum_{e \in node} P_e$ of the current node under analysis is lower with previously analysed, the algorithm explores branching nodes that can be originated from the last one in a way to explore that branch of the tree. Then, once the algorithm finds a node having a lower $\sum_{e \in node} P_e$, it checks if the $\bigcup_{S_i \in node}$ that originates, is equal to $\{1.....m\}$. If this is true, results are collected and i explore further nodes branching from the latter that i have just collected. Otherwise, i just explore further nodes.
At the end, the optimal solution so the set `G` $\subseteq$ `U` that expresses the following features,$\bigcup_{S_i \in G} = \{1.....m\}$ and $\sum_{e \in G} P_e$ is minimal, is provided. Also in this case e stands for an element of `U` present in the subset G.

To see our this algorithm should work, a pseudo-code has been developed. The final result counts of four functions, each performing a specific function:

1. the function `OverPass(node,n)`is responsible of bypassing a node, which represents one possible combination, in the moment in which the $\sum_{e \in node} P_e$ is higher with respect to one of the previously generated node;

2. the function `NextNode(node,n)` is responsible of building a node of size increased by one, so a possible subset of $\{1.....n\}$, originated from the current node under analysis. So it allows the exploration of novel nodes of higher size;

3. the function `ComputeSums(node,`$\{S_1.....S_n\}$`,`$\{P_1.....P_n\}$`)` is responsible to calculate the $\bigcup_{S_e \in node}$ and the $\sum_{e \in node} P_e$ of the current node so of the current subset of $\{1.....n\}$. Moreover it is responsible of eliminate possible repetitive elements from the $\bigcup_{S_e \in node}$ that may be originated by merging two sets of $\{S_1.....S_n\}$ carrying equal elements;

4. the function `BranchAndBound(n,m,`$\{S_1.....S_n\}$`,`$\{P_1.....P_n\}$`)` is responsible of checking the validity of the subset of $\{1.....n\}$ under study. The control is performed analysing the $\bigcup_{S_e \in node}$ that must be equal to $\{1.....m\}$ and the $\sum_{e \in node} P_e$ that must be minimal, where *node* is

the current subset of $\{1......n\}$.

```
OverPass(node,n)
  if a subset from which no further combinations are available then
    node ← remove the last element of the current node {go back to the vertex}
    node ← increase the last value of node by 1 {go to the node on the right}
    return node
  else
    node ← add 1 to the last element of the current node {bypass the current node}
    return node
  end if
```

This function, coupled with the function NextNode(node,n) is responsible of moving inside the internal nodes of the tree, where each node, as we said before, if a subsets of $\{1......n\}$. How does it works? Basically, in the moment in which the algorithm reaches a node that has $\sum_{e \in node} P_e$ that is higher with respect to the previously analyzed node, along the tree, actually there is no need of exploring branching nodes that may be generated from the current one. Therefore, this function skips the current branch of the tree and it moves the attention of the algorithm in evaluating the node that "stays" on the right side of the current one.

```
NextNode(node,n)

  if a subset from which no further combinations are available then
    node ← remove the last element of the current node {go back to the vertex}
    node ← increase the last value of node by 1 {go to right node}
    return node
  else
    node ← build node of size increased by 1 {explore lower level in the tree}
    return node
  end if
```

This function can be seen as an auxiliary function that coupled with the function OverPass(node,n), allows the algorithm to move inside the tree, analysing other nodes inside it. How does it works? Basically this function has the role of exploring higher sized nodes inside the tree in the moment in which the $\sum_{e \in node} P_e$ of the current node, is still minimal. This means that, branching points that may be originated from the current one, may still have an overall $\sum_{e \in node} P_e$ that is lower than the one of other nodes present in other branches of the tree. If the algorithm has reached a final leave of the tree, this function allows it to jump to the node on the right side of the branching node that has created the final leave where we arrived.

```
ComputeSums(node,{$S_1.....S_n$},{$P_1.....P_n$})

  sol ← empty array
  cost ← 0
  for all elements $e$ in node do
    sol ← $\sum_{e \in node} S_e$
    cost ← $\sum_{e \in node} P_e$
  end for
  LL ← empty array
  for all elements $i$ in sol do
    if elements $i$ not in LL then
      LL ← LL + i
    end if
  end for
  return LL sorted in increasing order, cost
```

The function ComputeSums(node,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$) performs two different functionalities. The first one is that it is entitle to calculate the $\bigcup_{S_e \in node}$ and the $\sum_{e \in node} P_e$ for the current node of the tree under analysis. The variable e stands for each element in the node. Basically, in the moment in which a new node of the tree is under analysis, we need to associate these two values to it in a way that the algorithm can discriminate among nodes having$\bigcup_{S_e \in node}$ equal to $\{1.....m\}$ and a $\sum_{e \in node} P_e$ to be minimal. So this function is actually very important. Regarding the second functionality, basically it is able to remove repetitive elements from the $\bigcup_{S_e \in node}$ that has been just created. The algorithm has to do this since if repetitive elements there might be no $\bigcup_{S_e \in node}$ being equal to $\{1.....m\}$ and so the algorithm won't produce the correct output.

```
BranchAndBound(n,m,{S_1.....S_n},{P_1.....P_n})

  $A \leftarrow \{1.....m\}$
  $minPrice \leftarrow$ inf
  $output \leftarrow$ empty array
  for $i \leftarrow 1$ to $n$ do
    $node \leftarrow$ array containing $i$
    $sol \leftarrow$ array containing the corresponding element at position $i$ in $S$
    $cost \leftarrow$ array containing the corresponding element at position $i$ in $P$
    while forever do
      if The current node is the last leaf of this section of the tree then
        stop while
      end if
      if cost $> minPrice$ then
        $node \leftarrow$ OverPass(node,n)
        $newSums \leftarrow$ ComputeSums(node,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$)
        $sol \leftarrow$ first element of newSums
        $cost \leftarrow$ second element of newSums
      else
        if sol = A then
          $minPrice \leftarrow$ cost
          $output \leftarrow$the current node under analysis so the subset of $\{1.....n\}$
        end if
        $node \leftarrow$ NextNode(node,n)
        $newSums \leftarrow$ ComputeSums(node,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$)
        $sol \leftarrow$ first element of newSums
        $cost \leftarrow$ second element of newSums
      end if
    end while
  end for
  return minCost,output
```

This function is the most important one of the algorithm. Basically it has the role of filtering those nodes in the tree and keep on track of those having the $\bigcup_{S_e \in node}$ being equal to $\{1.....m\}$ and having the $\sum_{e \in node} P_e$ minimal. In fact, it provides, at the end of the execution the optimal result. How does it works? Basically it starts from those sets having size equal to 1. From the first of them, it generates all possible internal nodes, branching from it, by means of the two functions that we have seen before OverPass(node,n) and NextNode(node,n). Remember that each node is a possible subset of $\{1.....n\}$. Then, per each node, the $\bigcup_{S_e \in node}$ and the $\sum_{e \in node} P_e$ is calculated by means of the function ComputeSums(node,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$): if the $\sum_{e \in node} P_e$ is not minimal at the current iteration step, there is no need of exploring branching nodes that may originate from the latter. So all relative nodes are skipped; otherwise, if the $\sum_{e \in node} P_e$ is minimal in the current iteration step, the algorithm checks if the $\bigcup_{S_e \in node}$ is equal to $\{1.....m\}$. If this is true, results, so the sum and the node are collected. Then we can continue in exploring this branch of the current node. At the end of the iteration, the algorithm will provide the optimal solution.

4 The implementation-Branch And Bound

In this section we present a possible implementation of the algorithm described above. The programming language that has been used is `python` version 3.7.
Looking at the psedo-code that has been designed, we observe the presence of three functions. So,also two functions were implemented in python:

This function is responsible of skipping the branching nodes of the tree that may originate from the current one since the $\sum_{e \in node} P_e$ is not minimal at the current iteration step.

```
def OverPass(node,n):
        if node[-1] == n-1:
                back = node[:-1]
                back[-1] += 1
                return back
        else:
                node[-1] += 1
                return node
```

This function is responsible of analyzing branching nodes that may originate from the current one since its $\sum_{e \in node} P_e$ is minimal at the current iteration step.

```
def NextNode(node,n):
        if node[-1] == n-1:
                back = node[:-1]
                back[-1] += 1
                return back
        else:
                node = node + [node[-1] +1]
                return node
```

This function is able to calculate the $\sum_{e \in node} P_e$ and the $\bigcup_{S_e \in node}$ where node *node* stands for the current subset of $\{1.....n\}$ under analysis. Then, it also removes repetitive elements from the $\bigcup_{S_e \in node}$ that has been just originated.

```
def ComputeSums(node, S, P):
        sol = []
        cost = 0
        for e in node:
                sol += S[e]
                cost += P[e]
        LL = []
        for i in sol:
                if i not in LL:
                        LL += [i,]
        return sorted(LL),cost
```

This function is entitled to explore the tree where each node is a subset of $\{1.....n\}$. The subsets are created along the iteration process in a way that at the end the optimal solution, so the subset G of $\{1......n\}$ having $\bigcup_{S_e \in G} = \{1.....m\}$ and $\sum_{e \in G} P_e$ is minimal, is provided.

```
import math

def BranchAndBound(n,m,S,P):
      A = [y for y in range(1,m+1)]
      minPrice = math.inf
      output = []
      for x in range(n-1):
            node = [x]
            sol = S[x]
            cost = P[x]
            while True:
                  if node == [x,n-1] or node == [n-1]:
                        break
                  if cost > minPrice:
                        node = OverPass(node,n)
                        newSums = ComputeSums(node,S,P)
                        sol = newSums[0]
                        cost = newSums[1]
                  else:
                        if sol == A:
                              minPrice = cost
                              output = node[:]
                        node = NextNode(node,n)
                        newSums = ComputeSums(node,S,P)
                        sol = newSums[0]
                        cost = newSums[1]

      return minPrice,output
```

5 The algorithm-greedy

Considering the algorithm that we have just developed, the branch and bound paradigm ensure us to reach the optimal solution for any instance values. This is due to the fact that, during the overall iteration process, all possible combinations of elements in the set $\{1.....n\}$ has been analyzed and the algorithm will select the one G having $\bigcup_{S_e \in G} = \{1.....m\}$ and $\sum_{e \in G} P_e$ is minimal. However, this strategy results to have an high time complexity since as the size of $\{1.....n\}$ increases, the number of combinations increases exponentially.
How can we improve the process? Basically, one can decide to develop the algorithm following a `greedy` strategy where, at each iteration step, solutions that are locally optimal are considered. Actually, at the end of the process, the final result may not be the optimal one since solutions that are locally optimal are not necessarily optimal at the global level. This strategy will reduce a lot the time complexity, at cost of the correctness since the algorithm results to be approximately correct.

To develop the greedy algorithm, a peculiar strategy has been applied. The latter is based on the calculation of the ratio "quantity-cost" or known as `cost effectiveness` for each element of $\{1.....n\}$ each associated to a value S_e from the set $\{S_1.....S_n\}$and to a value P_e from the set $\{P_1.....P_n\}$. Basically, the algorithm proceeds into evaluating how favorable is to add an element from $\{1.....n\}$ to the final solution in terms of cost and new elements from the set $\{1.....m\}$ that will be provided by the set of $\{S_1.....S_n\}$ associated to the element of $\{1.....n\}$ selected. In this way, the algorithm, at each iteration step, will select those elements of $\{1.....n\}$ that, locally, will provide the highest number of uncovered elements of the set $\{1.....m\}$ at lowest cost. This alternative strategy has been analyzed using some sources on the internet [3].

To see how this algorithm should work, a pseudo-code has been developed. The final result counts of three functions, each performing a specific function:

1. The function ClearResult($\{L_1.....L_p\}$) takes an array of integer numbers and it filters it removing all possible repetitive elements i.e if the array contains the following numbers $\{1, 1, 2, 2, 3, 4, 5\}$, this function will clear it providing the following array $\{1, 2, 3, 4, 5\}$;
2. The function effectiveness(c,$\{I_1.....I_x\}$,$\{LL_1.....LL_y\}$) is responsible of calculating the cost effectiveness of adding an element from $\{1.....n\}$ to the final solution. In fact, the function takes three arguments that are necessary to ensure this fact: c is the prize of an element e from $\{1.....n\}$; $\{LL_1.....LL_y\}$ is the set from $\{S_1.....S_n\}$ associated to the element e; $\{I_1.....I_x\}$ represents the current solution at each iteration set, so the set containing already covered elements of $\{1.....m\}$.
3. the function greedy((n,$\{1.....m\}$,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$) is responsible of producing the final result so the subset $G \subseteq U$ such that $\bigcup_{S_e \in G} = \{1.....m\}$ and having a price equal to $\sum_{e \in G} P_e$ that may not be necessarily minimal.

```
clearResult($\{L_1.....L_p\}$)
  sort elements in $\{L_1.....L_p\}$ in increasing order
  $LL \leftarrow$ empty array
  for $e \leftarrow 1$ to $p$ do
    if $L_e$ not in $LL$ then
      $LL \leftarrow LL + L_e$
    end if
  end for
  return LL
```

This function can be considered as a secondary function of the algorithm, even if its role is very important. Basically it is responsible of removing repetitive elements inside an array composed of integer numbers. Actually this process is quite important. In the moment in which the algorithm generates subset G of $\{1.....n\}$, it creates a $\bigcup_{S_e \in G}$ that merge some sets of $\{S_1.....S_n\}$ each associated to element e of G. So it can happen that some components of $\{S_1.....S_n\}$ may have common elements. Therefore, merging them in a unique set, may introduce repetitive components that can distort the search since any of the subsets of $\{1.....n\}$ won't respect the following requirement: $\bigcup_{S_e \in subset} = \{1.....m\}$.

```
effectiveness(c,$\{I_1.....I_x\}$,$\{LL_1.....LL_y\}$)
  $count \leftarrow 0$
  $cost \leftarrow 0$
  for $j \leftarrow 1$ to $LL_y$ do
    if $LL_j$ not in $\{I_1.....I_x\}$ then
      $count \leftarrow$ count + 1
    end if
  end for
  if count = 0 then
  $cost \leftarrow \infty$
  else
    $cost \leftarrow$ c/count
  end if
  return cost
```

This function is very important since it is responsible to calculate the cost effectiveness of adding each element from $\{1.....n\}$ to the final solution. Basically it uses a for loop to compare two sets: $\{I_1.....I_x\}$ which is the set counting the number of covered elements of $\{1.....m\}$. So we could

say that, during the iteration process, $\{I_1.....I_x\}$ is a subset of $\{1.....m\}$ but at the end of the procedure, the two set results to be the same; $\{LL_1.....LL_y\}$ which is simply the set of $\{S_1.....S_n\}$ associated to the current element of $\{1.....n\}$ for which we want to calculate the cost effectiveness. So, using this `for loop` the function is able to count the number of elements of $\{LL_1.....LL_y\}$ that has not been already covered by the set $\{I_1.....I_x\}$.
After this loop, the number of uncovered elements is evaluated: if it results to be 0 means that all elements of $\{LL_1.....LL_y\}$ are already covered by $\{I_1.....I_x\}$. So, there is no meaning of considering the current element of $\{1.....n\}$ since it won't provide any additional functionality. So the cost of considering this element is set to the maximum prize possible, so ∞.
Instead, if the result is not 0 means that some elements of LL has not been covered in the set I. For this reason, the algorithm calculates the ratio between the cost of the current element of $\{1.....n\}$ and the number of uncovered functionalities we found. The lower the ratio is, the higher will be the number of "novel" functionalities provided by the current package.

```
GREEDY((N,M,$\{S_1.....S_n\}$,$\{P_1.....P_n\}$)
  $packages \leftarrow \{1.....n\}$
  $A \leftarrow \{1.....m\}$
  $sol \leftarrow$ empty array
  $totPrice \leftarrow 0$
  [$pack \leftarrow$] empty array
  while sol $\neq A$ do
    $currentCE \leftarrow$ empty array
    for $i \leftarrow 1$ to $S_n$ do
      $cost \leftarrow$ EFFECTIVENESS($P_i$,$sol$,$S_i$)
      $currentCE \leftarrow$ currentCE + cost
    end for
    $lowerCostRatio \leftarrow$ position of the minimum value $M_e$ in currentCE
    $sol \leftarrow$ sol + $S_{lowerCostRatio}$
    $sol \leftarrow$ CLEARRESULT(SOL)
    $totPrice \leftarrow$ totPrice + $P_{lowerCostRatio}$
    $pack \leftarrow$ pack + $packages_{lowerCostRatio}$
end whilepack,totPrice
```

This function is responsible of providing the final result of the algorithm so the subset G of $\{1.....n\}$ that has the $\bigcup_{S_e \in G} = \{1.....m\}$ but since we said that the algorithm provides an approximate solution, the $\sum_{e \in G} P_e$ may not be minimal.
The work of the function is based on the execution of a `while loop` and of a nested `for loop`. Basically the while loop is executed a number of time until an array called *sol*, that has been initialized empty before, is equal to the array $\{1.....m\}$. Then, at each validation of the while loop, if it results to be TRUE, a for loop is executed. The latter is responsible of calculating the cost effectiveness of each set in the set$\{S_1.....S_n\}$ and, for each cost effectiveness calculated, the result is collected in and array called $currentCE$.
In the moment in which the for loop is over, the array $currentCE$ is composed of a total number of integers equal to the size of the set $\{S_1.....S_n\}$. So we basically have one cost effectiveness value per set. Then, among these, we select the one having the minimum value that will represent the element in $\{1.....n\}$ that is associated to the set of $\{S_1.....S_n\}$ that will provide the highest number of functionalities with the lowest price. In fact the set of $\{S_1.....S_n\}$ will be added to the empty array *sol* and after that, the algorithm removes possible repetitive elements from *sol* by calling the function CLEARRESULT(SOL).
Finally the function updates an array called *pack* with the element of $\{1.....n\}$ associated to the set of $\{S_1.....S_n\}$ that we have just added to *sol* and an other variable called *totPrice* adding the price of the element that we have just added to the variable *pack*.
The execution proceeds until the guard of the while loop fails, meaning that we have covered all elements in $\{1.....m\}$.

6 The Implementation-greedy

In this section we present a possible implementation of the algorithm described above. The programming language that has been used is `python` version 3.7.
Looking at the psedo-code that has been designed, we observe the presence of three functions. So,also three functions were implemented in python:

This function is responsible of eliminating possible repetitive elements of $\bigcup_{S_e \in G}$ of the current node under analysis called G that may have originated from merging elements of $\{S_1......S_n\}$ having some equal elements

```
def clearResult(L):
       L = sorted(L)
       LL = []
       for e in L:
              if e not in LL:
                     LL += [e,]
       return LL
```

This function is responsible of calculating the cost effectiveness of any set of $\{S_1.....S_n\}$. Basically, if the number of uncovered elements of a set is high, the cost effectiveness results to be low since adding this set to the final solution will result into covering an higher number of elements with the lowest possible cost. Otherwise, if the cost effectiveness is high, like when the number of uncovered elements is 0, the cost effectiveness will result to be higher.

```
def effectiveness(c,I,LL):
       count = 0
       cost = 0
       for e in LL:
              if e not in I:
                     count += 1
       if count == 0:
              cost = math.inf
              return cost
       else:
              cost = c/count
              return cost
```

This function is responsible of finding the subset G of $\{1.....n\}$ such that $\bigcup_{S_i \in G} = \{1.....m\}$ but the price may not be minimal since we are performing a greedy strategy to solve the problem.

```
import math

def greedy(n,m,S,P):
       packages = [x for x in range(n)]
       A = [y for y in range(1,m+1)]
       sol = []
       totPrice = 0
       pack = []
       while sol != A:
              currentCE = []
              for i in S:
                     cost = effectiveness(P[S.index(i)],sol,i)
                     currentCE += [cost,]

              lowerCostRatio = currentCE.index(min(currentCE))
              sol += S[lowerCostRatio]
```

```
            sol = clearResult(sol)
            totPrice += P[lowerCostRatio]
            pack += [packages[lowerCostRatio],]

    return totPrice,sorted(pack)
```

7 The approximation ratio

Considering that an approximation algorithm has been developed, we need to understand how the solution it provides is far from the exact one. So looking at different sources [2], they confirmed that the approximation ratio of this algorithm is equal to the harmonic sum, H_m which can be approximated to $log(m)$, where m is the number of elements in $\{1.....m\}$. The harmonic sum is the sum of reciprocals of the positive integers.
To prove this, we based on the different sources that has been analyzed:

`Theorem`:the approximation ratio of the algorithm results to be H_m that can be approximated to $log(m)$

`Proof`
To prove this theorem we need to first recall which are the class of instances on which the algorithm is base:

1. a set `U` equal to $\{1.....n\}$;
2. a universe set called `A` that is equal to $\{1.....m\}$;
3. a set `S` equal to $\{S_1.....S_n\}$ where, each element S_i is a subset of $\{1.....m\}$;
4. a set `P` containing $\{P_1.....P_n\}$ elements.

Then consider these two following statements:

- the cost of set G equal to $\{1.....j\}$ provided by the greedy as solution, is equal to $\sum_{i=1}^{j} P_i$, where i represents each element of G that is a subset of $\{1.....n\}$. To each element i a price value P_i and a set S_i are associated;
- what we want to see if that the cost of adding the k^{th} element to a possible solution, has the lowest cost effectiveness among all possible, meaning that the price of elements of $\{1.....n\}$ to which a set of $\{S_1.....S_m\}$ is lower or equal to the ratio between the cost of the optimal solution over the number of uncovered elements in the current solution. More formally:

$$P_k \leq \frac{OPT}{m-k+1}$$

Now let's consider two sets: `I` equal to $\{I_1.....I_{k-1}\}$ which represents the set of covered elements of $\{1.....m\}$ up to the element k and the set `O` equal to $\{O_1.....O_p\}$ such that $\bigcup_{S_i \in O}$ is the optimal solution so the one having the $\sum_{e \in O} P_e$ minimal.
Therefore, if the set `I` contains covered elements up to k means that the number of uncovered elements is equal to $|A - I|$.
Where do i pick the reaming elements such that `I` becomes equal to `A`? Basically uncovered elements are at most intersections of the optimal set and the uncovered set that contains the element you are looking for.

So, to better understand, the elements can be picked from these intersections sets:

$$|O_1 \cap (A-I)| + |O_2 \cap (A-I)| + ... + |O_p \cap (A-I)|$$

Which of these intersections set do we have to consider? Basically in the greedy strategy we calculate the cost effectiveness of each of them therefore we have to select the intersection set providing the highest number of uncovered elements with the lowest cost. So we have to select the intersection set having the minimal cost effectiveness value.

Now, to formalize this concept we need to set a variable called CE representing the cost effectiveness. The value of CE must be such to be:

$$CE \leq \frac{P_i}{|O_i \cap (A-I)|}$$

The smallest cost effectiveness can be either smaller or equal to the optimal set.
Now, after applying some algebra we end with this result:

$$P_i \leq CE * |O_i \cap (A-I)|$$

Now, considering the variable P_i, what is its value? It basically represents the optimal value for the price of the optimal solution, so its value is: $\sum_{i \in O} P_i$. Lets call this variable as OPT.
Now, making some algebra calculations we have:

$$\sum_{i \in O} P_i \leq CE * \sum_{i \in O} |O_i \cap (A-I)|$$

On the left member of the inequality, the value results to be the value OPT; while, on the right member of the inequality, it results to be equal to $|A-I|$ which is the number of uncovered elements. This is due to the fact that if I merge all elements in $\{O_1 .... . O_p\}$ i obtain the optimal set that collect a number of elements that makes it equal to `A`. Therefore, if i intersect two sets that are identical, the resulting intersection set results to be empty. So in our case, the number of elements present in the intersection set is equal to the number of uncovered elements so $|A-I|$. Then, proceeding in making some algebra, we end with the following inequality:

$$CE \leq \frac{OPT}{|A-I|}$$

So the cost effectiveness of adding the k^{th} element is equal to:

$$CE \leq \frac{OPT}{m-k+1}$$

Now, putting together the two statements that we introduced before the prof of the theorem, we can show that $P_k \leq \frac{OPT}{m-k+1}$, we can conclude that:

$$\sum_{k=1}^{m} P_k \leq \sum_{k=1}^{m} \frac{OPT}{m-k+1}$$

$$APP \leq OPT * \left(1 + \frac{1}{2} + \cdots + \frac{1}{m}\right)$$

$$APP \leq OPT * log(m)$$

$$\textit{Approximation ratio}:\ APP/OPT = OPT * log(m)/OPT = log(m)$$

So we can conclude the approximate solution provided by the algorithm results to be $log(m)$ distant form the optimal one since, it can be obtained by multiplying the value of the optimal solution with $log(m)$.

Now we have to prove that the approximation ratio of our algorithm for a given set of instances is at most equal or lower than $log(m)$. The algorithm has been tested on five instances each having a different value of `n`, `m` $\{S_1 .... . S_n\}$ and $\{P_1 .... . P_n\}$ both on exact and greedy algorithms. Since we need to see how the the cost of the greedy solution is distant from the one provided by the optimal, we have to calculate the ratio between them, seen as ratio between approximate solution over optimal one, and see if the value obtained is lower or equal to $log(m)$.

$$greedy\ sol \leq optimal\ sol * log(m)$$
$$\frac{greedysol}{optimalsol} \leq log(m)$$

1. `INSTANCE NUMBER ONE`: the value of `m` is 5. The exact solution provides a cost equal to 23 while the greedy one has a value of 23. In this case the exact and the greedy solution, coincide;
2. `INSTANCE NUMBER TWO`: the value of `m` is 15. The exact solution provides a cost equal to 20, while the greedy one provides 21. Their ratio is equal to 1.05, while the log(15) is equal to 2.7080. Therefore we see that $\frac{greedy}{optimal} \leq log(m)$;
3. `INSTANCE NUMBER THREE`: the value of `m` is 5. The exact solution provides a cost equal to 16, while the greedy one has a value of 16. In this case, the exact and the greedy solutions coincide;
4. `INSTANCE NUMBER FOUR`: the value of `m` is 40. The exact solution provides a cost equal to 106 while the greedy one provides 122. Their ratio is equal to 1.1509 and the log(40) is 3.6888. Therefore we see that $\frac{greedy}{optimal} \leq log(m)$;
5. `INSTANCE NUMBER FIVE`: the value of `m` is equal to 30. The exact solution provides a cost equal to 138 while the greedy one provides 138. In this case, the exact and the greedy solution coincide.

8 Performance evaluation

In the following section, we show the evaluation of the performance of the two algorithms that has been developed and implemented. The performance has been evaluated considering how the time needed by the algorithms, running them on five different instances groups each having a different value of `n`,`m`,$\{S_1......S_n\}$ and $\{P_1.....P_n\}$. Time has been measured in seconds. The data cornering time, has been extrapolated by testing the algorithms using the python package `cProfile`.

`INSTANCE NUMBER ONE`

- value of `n`:10
- solution cost value: exact=23; greedy=23
- solution set value: exact=[2,4,9]; greedy=[2,4,9]
- seconds: exact=0.001; greedy=0.000

`INSTANCE NUMBER TWO`

- value of `n`:45
- solution cost value: exact=20; greedy=21
- solution set value: exact=[1,39]; greedy=[1,11,7]
- seconds: exact=0.046; greedy=0.000

`INSTANCE NUMBER THREE`

- value of `n`:21
- solution cost value: exact=16; exact=16
- solution set value: exact=[4,14,17]; greedy=[4,14,17]

- seconds: exact=0.002; greedy=0.000

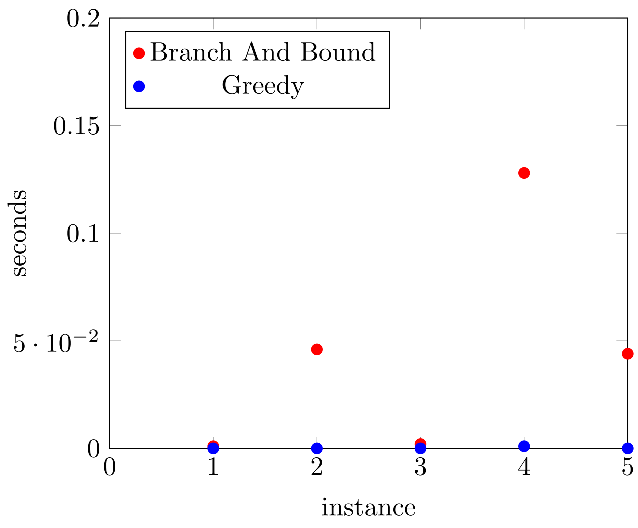
`INSTANCE NUMBER FOUR`

- value of `n`:23
- solution cost value: exact=106; greedy=122
- solution set value: exact=[11,14,18]; greedy=[9,14,18,22]
- seconds: exact=0.128; greedy=0.001

`INSTANCE NUMBER FIVE`

- value of `n`:23
- solution cost value: exact=138; greedy=138
- solution set value: exact=[6,15,16]; greedy=[6,15,16]
- seconds: exact=0.044; greedy=0.000

These results has been plotted in a graph where, the `X-axis` host the number of the instance that we are analyzing while the `Y-axis` is used to report the number of seconds needed by the exact and by the greedy solution in each instance kind that has been considered.



9 Reference Literature

Considering that different sources has been used to develop the previous algorithms, we cite them out in this section of the document.

References

[1] Prof. Nicole Megow (summer 2017) Approximation Algorithms. Universitt Bremen. Link: https://cslog.uni-bremen.de/teaching/summer17/approx-algorithms/resource/lec3.pdf

[2] Tamara Stern (2006) Seminar in Theoretical Computer Science. Link: http://math.mit.edu/ goemans/18434S06/setcover-tamara.pdf

[3] GeeksforGooks-a computer science portal for geeks-Set Cover Problem. Link: https://www.geeksforgeeks.org/set-cover-problem-set-1-greedy-approximate-algorithm/