Assignment 2

Andrea Rubbi

May 2019

1 The 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 Pi Euros, and offering some functionalities: unfortunately, not all packages are equivalent! 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.

1.1 Mathematical description of the problem

The problem can be formulated as follows: minimize $\sum_{set \epsilon S} W_{\text{set}} * x_{\text{set}}$ subject to $\sum_{set:e \epsilon S} x_{\text{set}} \geq 1$. Where $x_{\text{set}} \epsilon$ {0,1}, hence a set is present or not in the covering subset. 'S' is the covering set, 'W' is the list of set costs, 'U' is the universe and 'e' is an element of the universe. So the problem requires a minimization of the total cost taking into account that every element 'e' must be covered at least once.

2 Greedy algorithm

2.1 Pseudocode

```
1   SetCover(Universe,Subsets,Costs)
2       cost ← 0
3       elements ← all elements in subsets taken just one time
4       if elements != universe:
5           return
6       covered ← []
7       cover ← []
8       while covered != elements:
9           subset ← set with the highest 'elements not in covered'/'cost' ratio
10            cover ← cover + subset
11            cost ← cost + cost of subset
12            covered ← covered + element in subset and not in covered
13      return cover,cost
```

2.2 Explanation of the algorithm

The main idea of the algorithm is to cover the universe taking every time the apparently most convenient set in the sets list. In other words: every while cycle the program will search among all sets and will take the one with the highest ratio between the elements not yet covered and the relative cost of the set. This algorithm doesn't always give the best result, but certainly it gives an optimal one.

2.3 Implementation of the algorithm

```
__author__ = "Andrea Rubbi"

def set_cover(universe, subsets,costs):
    cost=0
    elements = set(e for s in subsets for e in s)
    if elements != universe:
        return None
    covered = set()
    cover = []
    while covered != elements:
        subset = max(subsets,
        key=lambda s: len(s - covered)/costs[subsets.index(s)])
        cover.append(subset)
        cost+=costs[subsets.index(subset)]
        covered |= subset

    return cover, cost

def main():
    universe = set(range(1, 41))
    sub = [[1, 3, 4, 6, 7, 9, 10, 15, 16, 18, 26, 31, 32, 35, 36, 38, 39, 40], [1, 2, 3

    subsets = [set(x) for x in sub]
    costs =  [59, 68, 56, 50, 75, 95, 71, 66, 30, 28, 42, 50, 68, 34, 29, 52, 70, 85, 2
    cover = set_cover(universe, subsets,costs)
    print('covering sets= ',cover[0],'\n',
          'cost= ',cover[1],'$')

if __name__ == '__main__':
    main()
```

3 Branch and Bound algorithm

3.1 Pseudocode

```
1   SetCover(universe,sets,costs)
2       subset ← [1]* |sets|
3       subset_1 ← 0
4       bestcost ← sum(costs)
5       i ← 2
6       while i > 0:
7           if i < |sets|:
8               cost ← 0
9               tset ← []
10               for k ← 1 to i:
11                   cost ← cost + subset_k * costs_k
12                   if subset_k = 1:
13                       tset ← tset + sets_k
14               if cost > bestcost :
15                   subset,i ← bypassbranch(subset,i)
16               else:
17                   subset,i ← nextvertex(subset,i,|sets|)
18           else:
19               cost ← 0
20               fset ← []
21               for k ← 1 to i:
22                   cost ← cost + subset_k * costs_k
23                   if subset_k = 1:
24                       fset ← fset + sets_k
25                   if cost < bestcost and elements in fset = universe:
26                       bestcost ← cost
27                       bestsubset ← subset
28               subset,i ← nextvertex(subset,i,|sets|)
29       return bestcost,bestsubset

1   bypassbranch(subset,i)
2       for j ← i to 1
3           if subset_j = 0:
4               subset_j ← 1
5               return subset,j+1
6       return subset,0

1   nextvertex(subset,i,m)
2       if i < m:
3           subset_i ← 0
4           return subset,i+1
5       else:
6           for j ← m to 1
7               if subset_j = 0:
8                   subset_j ← 1
9                   return subset,j+1
10      return subset,0
```

3.2 Explanation of the algorithm

This is a branch and bound algorithm where branches are developed as long as their vertexes don't became greater than the best cost found so far and the elements in selected sets form the whole universe. There are two auxiliary functions: bypass branch, that simply bypasses a not feasible or not optimal solution; and next vertex, essential to keep on creating new subsets. This algorithm gives always the best solution because it compares all possible solutions and takes the one that has the lowest total cost.

3.3 Implementation of the algorithm

```
__author__ = "Andrea Rubbi"
import time

def bypassbranch(subset, i):#bypass a branch
    for j in range(i-1, -1, -1):
        if subset[j] == 0:
            subset[j] = 1
            return subset, j+1

    return subset, 0

def nextvertex(subset, i, m):
    if i < m:
        subset[i] = 0
        return subset, i+1
    else:
        for j in range(m-1, -1, -1):
            if subset[j] == 0:
                subset[j] = 1
                return subset, j+1

    return subset, 0

def BB(universe,sets,costs):
    subset = [1 for x in range(len(sets))]#all sets in
    subset[0] = 0
    bestCost = sum(costs) #actually the worst cost
    i = 1

    while i > 0:

        if i < len(sets):
            cost, tSet = 0, set()# t for temporary
            for k in range(i):
                cost += subset[k]*costs[k]#if 1 adds the cost to total
                if subset[k] == 1: tSet.update(set(sets[k]))#if 1 add the set to the co

            if cost > bestCost:#if the cost is larger than the currently best one, no n
                subset, i = bypassbranch(subset, i)
                continue
            for k in range(i, len(sets)): tSet.update(set(sets[k]))
            if tSet != universe:#that means that the set was essential at this point to
                subset, i = bypassbranch(subset, i)
```

```
            else:
                subset, i = nextvertex(subset, i, len(sets))

        else:
            cost, fSet = 0, set()# f for final
            for k in range(i):
                cost += subset[k]*costs[k]
                if subset[k] == 1: fSet.update(set(sets[k]))

            if cost < bestCost and fSet == universe:
                bestCost = cost
                bestSubset = subset[:]
            subset, i = nextvertex(subset, i , len(sets))

    return bestCost, bestSubset

def main(a,b,c,z=time.time()):
    m = a
    S = b
    C = c
    F = set([x for x in range(1,m+1)])
    X=(BB(F,S,C))
    cost= X[0]
    sets= X[1]
    cover= []
    for x in range(len(sets)):
        if sets[x]==1:
            cover.append(S[x])
    print('covering sets: ',cover,'\n','total cost: ',cost,'$')
    print('time:',time.time()-z)

m1= 5
S1 = [[1,3],[2],[1,2,5],[3,5],[4],[5],[1,3],[2,4,5],[1,2],[2,3]]
P1 = [11,4,9,12,5,4,13,12,8,9]
m2 = 15
S2 = [[2, 7, 8, 10, 12, 13], [1, 3, 5, 8, 10, 11, 12, 15], [1, 2, 3, 4, 5, 6, 7, 12, 13
P2 = [16, 7, 16, 39, 29, 35, 19, 27, 27, 33, 38, 8, 41, 16, 12, 7, 41, 6, 34, 48, 23, 1
m3 = 5
S3 = [[1], [1, 2, 3, 4, 5], [2, 3], [2, 3, 4, 5], [1, 3, 4], [5], [1, 2, 4], [1, 3, 4,
P3 = [44, 44, 39, 24, 5, 30, 26, 42, 28, 12, 6, 45, 37, 33, 5, 42, 26, 6, 38, 11, 28]
m4 = 40
S4 = [[1, 3, 4, 6, 7, 9, 10, 15, 16, 18, 26, 31, 32, 35, 36, 38, 39, 40], [1, 2, 3, 4,
P4 = [59, 68, 56, 50, 75, 95, 71, 66, 30, 28, 42, 50, 68, 34, 29, 52, 70, 85, 27, 40, 7
m5 = 30
n5 = 23
S5 = [[2, 3, 4, 8, 9, 10, 11, 12, 15, 16, 18, 19, 22, 23, 24, 26, 27, 28, 29], [1, 2, 4
P5 = [60, 79, 49, 65, 88, 83, 38, 44, 54, 100, 65, 53, 43, 73, 63, 35, 65, 92, 74, 79, 6
if __name__=='__main__':
    main(m1,S1,P1)
    main(m2,S2,P2)
    main(m3,S3,P3)
    main(m4,S4,P4)
    main(m5,S5,P5)
```

4 Comparison of the two algorithms

4.1 Approximation ratio

4.1.1 Theoretical

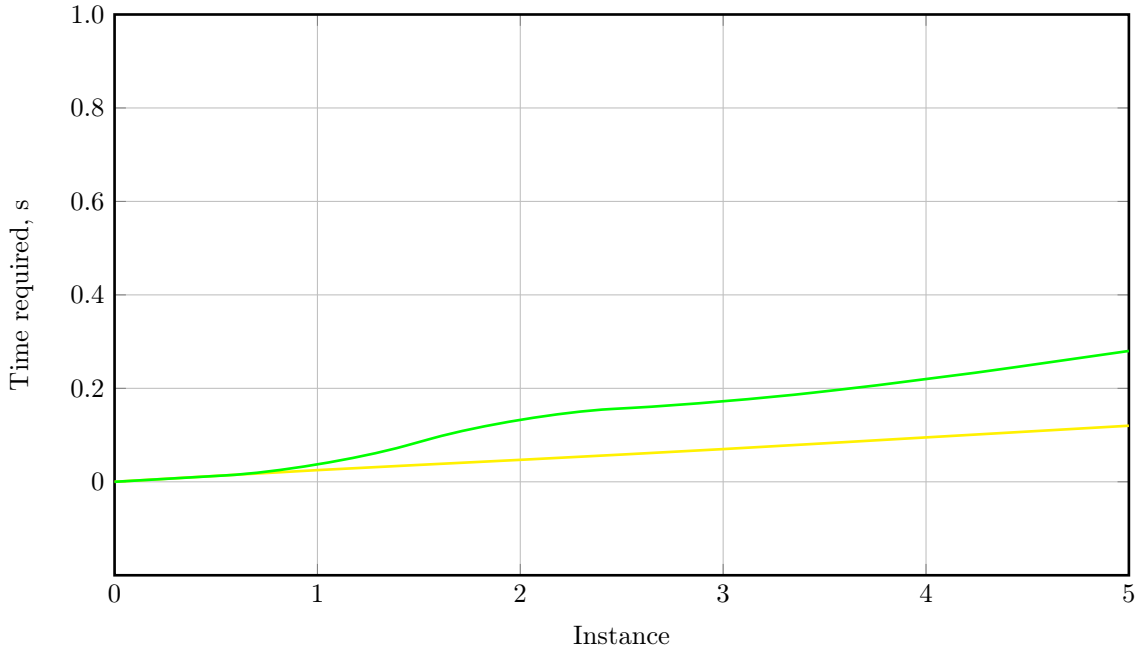
The Greedy algorithm is an (ln(n) + 1) approximation algorithm for weighted set cover problem. (source: https://cslog.uni-bremen.de/teaching/summer17/approx-algorithms/resource/lec3.pdf) (NB: this is also the source of the greedy algorithm idea)

4.1.2 Empirical

Instance-n	BB cost	Greedy cost	Ratio
Instance-1	23	23	1,00
Instance-2	20	21	1,05
Instance-3	16	16	1,00
Instance-4	106	122	1,15
Instance-5	138	138	1,00

Since the ratio is equal to 1 or a little more, it's assumable that the greedy algorithm gives a good approximation, certainly within the expected (theoretical) bounds.

4.2 Running time



The yellow curve is relative to the greedy algorithm while the green one to the branch and bound algorithm.This last one is certainly slower, however it has good performances with these instances.