DADES DEL PROJECTE

Títol del projecte : Jutge.org : Heuristics and Integer Linear Programming for identifying common fail conditions of submitted solutions

Nom de l'estudiant : Divya Venkataramani

Titulació : Enginyeria Informàtica

Credits : 37.5

Director : Jordi Petit Silvestre

Department : Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL (nom i signatura)

President : Amalia Duch Brown

Vocal : Jordi Quer Boson

Secretari : Jordi Petit Silvestre

QUALIFICACIÓ

Qualificació :

Qualificació descriptiva :

Data :

TABLE OF CONTENTS

CHAPTER 1 : PROJECT BACKGROUND

1.1 Introduction :

jutge.org (https://www.jutge.org/)[1] is an open access educational online programming judge where students can try to solve more than 1000 problems using 22 programming languages. It has been functioning since 2006 and in these six years it has received hundreds of submissions for each of its problems.

A good programming course must involve a lot of coding practice. The conventional 'write-and- verify' method implemented in the form of written tests has proven to be both inefficient and arduous. Of all online programming tools, jutge.org is unique in the way that it has been designed as a code-practicing and testing environment aiding students and instructors.

An inherent problem with automated verification is the enforcement of leniency which is expected in the code-practicing phase.

"Any programming problem can be solved by adding a level of indirection."

In order to assist the students whose submissions were not accepted, we envisioned an automated response system that sends them an appropriate message informing them of the fail condition that they had ignored. This message is a 'distilled' set of inputs that holds a clue as to where the submitted program might have gone wrong. We derive information to create the 'distilled' set from the pile of failed submissions that have been accumulated over the years.

For every problem, we have a public input set and a private input set. The public input set is provided as sample test cases to the students. The private input set is used to verify the submissions. The submitted codes are executed in a sandbox environment to ensure the security and integrity of the system.

First we execute the submissions with the public inputs and eliminate the submissions that fail on any of the sample tests. This is because of two reasons. It helps in filtering out irrelevant programs which would have otherwise been misleading. It makes sense that we ignore the sample test cases as they are already made available to the students. Then we execute the remaining submissions with the private inputs. Every private input is associated with a set of submissions that fail on that input. A set cover program[2] is executed on the private inputs to arrive at the reduced set of inputs. The set cover algorithm is one of Karp's 21 NP-Complete problems. We discuss this concept in detail in Chapter 2.

The 'distilled' set is the minimal set of inputs that contain at least one input that

fails every incorrect submission that passes the public test.

```
for every submission in submissions:
    fail[submission]=0
    for every input in distilled_inputs:
        run submission with input
        if submission fails on input:
            fail[submission]=1
            break
```

When all the submissions are run on the distilled input set, all the incorrect submissions have their fail flag set to 1. Here, by all submissions we mean the submissions that were involved in the set cover algorithm. When a student, undergoing practice, submits a program that fails, he can. When asked for the hint, we can give him the distilled list or we can consult the distilled list corresponding to the problem, find the most significant input on the list that the program fails on and prompt it to the learner.

When a new submission arrives which fails on the private set, but happens to pass all the 'distilled' inputs, 'distilled' set needs to be refreshed. Now, the 'distilled' inputs does not help that student. But when the association between the inputs and the submissions that fail on each input are refreshed and the set cover algorithm is run again, the 'distilled' set becomes up-to-date. Now it has at least one element that applies to the new submission. This means that the 'distilled' set needs to be regenerated from time-to-time. This also emphasizes the need to store the associations for reuse. This also means that we need a diverse set of submissions to start with.

1.2 Goals of the project :

Objective:

To design an efficient, modular method to find the 'distilled' set of inputs for problems in jutge.org[1].

Tasks:

- To study the repository of problems in jutge.org[1]. The problems need to be classified based on the programming language, the layout of inputs and outputs, etc.
- To prepare a secure environment for compiling the user programs. Any submission should be treated like a malicious attack and must be jailed in a sand box environment. It includes switching to a low-privileged nobody user and

limiting the file system access. We will have to devise a method to compile submissions of a problem in bulk.

- To study the set cover problem and its various existing implementations. We intend to define various versions of set cover problem customized for our situation.
- To identify efficient data structures for holding the associations between the inputs and the submissions that fail on each input. We also need to determine the manner in which we are going to store this information for reuse.
- To overcome the differences and irregularities in the existing problems that represent 'loopholes' in the system. We will have to deal with different classes of input and output layouts. There might be occasional cases where the inputs do not adhere to its own blueprint.
- To analyze and compare the different implementations based on complexity and to identify implementations that favors specific types of problems. Once we have the solution in hand, we need to optimize it. In-depth analysis in terms of time and space complexity needs to be done.
- To identify effective means of code reuse such as incorporating the linux diff command.

1.3 Classification of problems :

We intend to target the various classes of problems in jutge.org[1]. First step towards realizing this task is to identify the bases of classification.

1.3.1 Language oriented :

The specifications of some problems are in different natural languages. The public and private test cases are provided separately for each language. The user programs that are submitted for different language versions of a problem are held in different directories.

In most cases, the problem is simply stated in different languages but the input and outputs do not depend on the language.

Let us consider two different cases of problems where the structures vary based on how much they depend on the language. Consider the problem, P21459 with the title '50 * 50 !=250'. The sample input and the sample output are as shown below.

Sample Input	Sample Output
50 50 002 003 9999 9999	1 2 2 0 0 4

In the above case the inputs and the outputs are not related to the language. This is the case in which we can reduce redundancy by storing the inputs in just one place for all the language versions.

Consider the problem P89867 with the title 'Classification of sequences (2)'. In this problem the output depends on the language.

Sample Input	Sample Output
2 3 5 7 10 0 -5 4	Strictly increasing Strictly decreasing Uniform

The problem is available in English and Catalan.

Sample Input	Sample Output
2 3 5 7 10 0 -5 4	Estrictament creixent Estrictament decreixent Uniforme

In this case, the input and the output cases need to be stored separately.

1.3.2 Delineated outputs :

The second classification of problems is based on the blueprint of the outputs. This information is essential when we need to split the outputs and match them with the corresponding correct outputs in order to verify them. Broadly, the problems can be classified into single-line inputs (outputs) and multi-line inputs (outputs).

The problem 'P33839' called 'Sum of digits' has single-line inputs and single-line

outputs. When given a number (in a single line) it returns a string stating the number of digits in the number in a single line. The sample test cases are shown below.

Sample Input	Sample Output
29 7 0 1020	The sum of the digits of 29 is 11 The sum of the digits of 7 is 7 The sum of the digits of 0 is 0 The sum of the digits of 1020 is 3

Now, let us have a look on the problem where an output consists of multiple lines. For this we take the problem 'P87812' where the style of the inputs and the outputs are in the format shown below.

Sample Input	Sample Output
9 3 19 18446744073709551614 1 10	3931067935 EMPTY DESK 4132970100 4208990443 4208990443 EMPTY DESK

Here the outputs are delimited by empty lines. The implementations have been designed to tackle more complicated cases which are yet to be encountered.

1.4 Structure of the project :

Here we discuss the directory structure of problems. We have chosen to work with a directory with path " /jutge/" as our root directory. We place all problem directories within the root. Below we have provided the tree structure of the root directory.

Directories and files pertaining to various problems can co-exist in the root

directory. But for the sake of explanation we have considered the structure of problem P46903 in particular.

Initially we have a couple of directories for each language version of each problem. In specific, one directory with “.pbm” in its name and another directory with “.sub” in its name. If the inputs and outputs are language-independent, then we can have just one “.pbm” directory in common for all “.sub” directories.

The “.pbm” directory contains several pairs of files that have the “.inp” or “.cor” in their file names. They represent input files and their corresponding correct output files. They contain a set of test input cases which can be public or private. Typically there is a file by name “sample.inp” with its correct outputs in a file by name “sample.cor”, both of which are publicly available. The rest of the files are private. In general, we also have a “buit.inp”-”buit.cor” pair which tests the programs with empty string as the input. We include the “buit” condition in the “distilled” set of all problems, wherever relevant, and neglect it during set cover process in order to avoid the “masking” effect it has on other significant private inputs.

The “.sub” directory has the user programs. This programs can be in any of the 22 programming languages supported by jutge.org. It has been noticed that C++ submissions form the majority of the submissions. The implementations therefore target “.cc” files but they can be easily adapted for any number of languages.

We have devised the following modular approach for our solution. It consists of the following python modules.

1. Compiling the programs (compile.py)
2. Using diff (breakinp.py)
3. Creating dictionary (createdict.py)
4. Applying set cover method using ILP(setcover.py)

The user programs in “.sub” directory are compiled and the resulting executables are stored in the “.cmp” directory. The rest of the files presented in the tree structure along with the python modules will be discussed in the following chapters.

```
/jutge/
├── breakinp.py
├── compile.py
├── createdict.py
├── P46903_args
├── P46903_ca.cmp
│   ├── fa47640e46a8bf871adaf1047bfa08a6-PE.exe
│   └── ffcea22dd513c6bbdb7ff2247caff025-PE.exe
├── P46903_ca.sub
│   ├── fa47640e46a8bf871adaf1047bfa08a6-PE.cc
│   └── ffcea22dd513c6bbdb7ff2247caff025-PE.cc
├── P46903_en.cmp
│   ├── fa9133630bf0c42e3581a7c5aac12288-EE.exe
│   ├── fc447caebdbe84e5e99630ee0f10e7fa-PE.exe
├── P46903_en.sub
│   ├── fa9133630bf0c42e3581a7c5aac12288-EE.cc
│   ├── fc447caebdbe84e5e99630ee0f10e7fa-PE.cc
├── P46903.pbm
│   ├── buit.cor
│   ├── buit.inp
│   ├── ma.cor
│   ├── ma.inp
│   ├── ma_inpcor
│   │   ├── 0.cor
│   │   ├── 0.inp
│   │   ├── 1.cor
│   │   ├── 1.inp
│   │   ├── 2.cor
│   │   ├── 2.inp
│   │   └── 3.cor
│   ├── random.cor
│   ├── random.inp
│   ├── random_inpcor
│   │   ├── 0.cor
│   │   ├── 0.inp
│   │   ├── 1.cor
│   │   ├── 1.inp
│   │   ├── 2.cor
│   │   ├── 2.inp
│   │   └── 3.cor
│   ├── sample.cor
│   ├── sample.inp
│   ├── sample_inpcor
│   │   ├── 0.cor
│   │   ├── 0.inp
│   │   ├── 1.cor
│   │   ├── 1.inp
│   │   └── 2.cor
│   ├── solution.cc
│   └── solution.exe
├── P46903_vinga_dict_break
├── P46903_vinga_distilled
├── P46903_vinga_time
└── setcover.py
```

CHAPTER 2 : SET COVER PROBLEM

2.1 Problem:

The set cover problem[2] can be explained as following. Consider the situation where

Universal set $U = \{E_1, E_2, E_3, ..., E_n\}$

Set of Subsets $S = \{S_1, S_2, S_3, ..., S_m\}$

The goal of the problem is to find a set, say I in the following way and I should be minimum.

$I = \{1,2,..., m\}$ such that $\bigcup_{i \in I} S_i = U$

In our context,

Universal set is the set of all incorrect submissions that pass the public test cases.

Each subset, S_i is the set of submissions that fail on private input, i.

The goal is to find the subset of private inputs, I, such that $\bigcup_{i \in I} S_i = U$

I is called the “distilled” set of inputs.

The idea of the problem is that each element of the universal set is incident to at least one of the subsets. The aim is to find minimum number of subsets, whose union is equal to the universal set. Basically, the attempt is to cover all the elements in the universal set and hence the name. Additionally, the number of the subsets should be minimized. It has been proven that the set cover problem is NP-Complete. This complexity class is explained in the third section of this chapter.

2.2 Example :

Consider an example of set cover problem. The sets of all elements are represented as

U={“abc.exe”, “def.exe”, “ghi.exe”, “jkl.exe”, “mno.exe”}

The private input set ={1,2,3}

Set of submissions that fail on the input,1 = S_1 = {“def.exe”, “jkl.exe”}

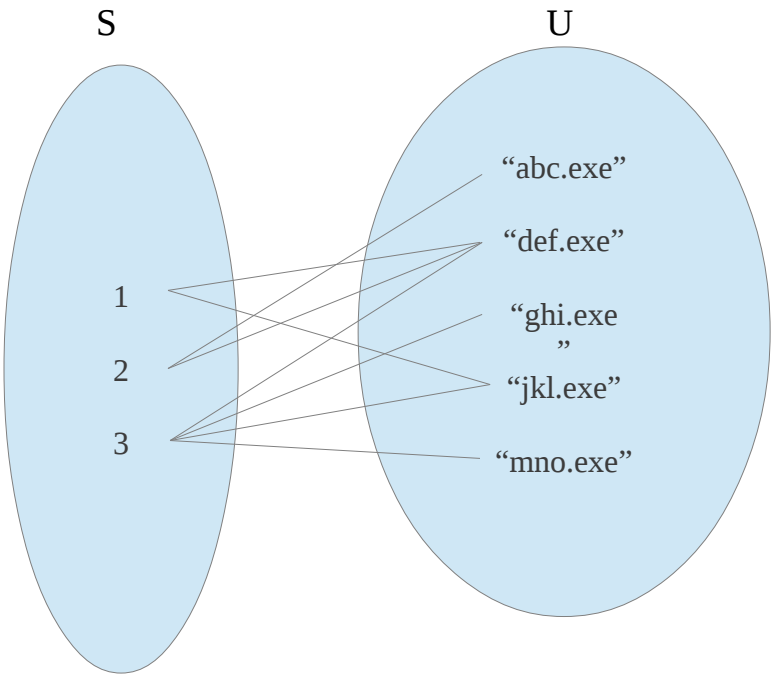
Set of submissions that fail on the input,2 = S_2 = {“abc.exe”, “def.exe”}

Set of submissions that fail on the input,3 = S_3 = { “def.exe”, “ghi.exe”, “jkl.exe”, “mno.exe”}

$S=\{S_1, S_2, S_3\}$

It can be inferred from the sets that the distilled set for this imaginary problem would be $I = \{ S_2, S_3\}$

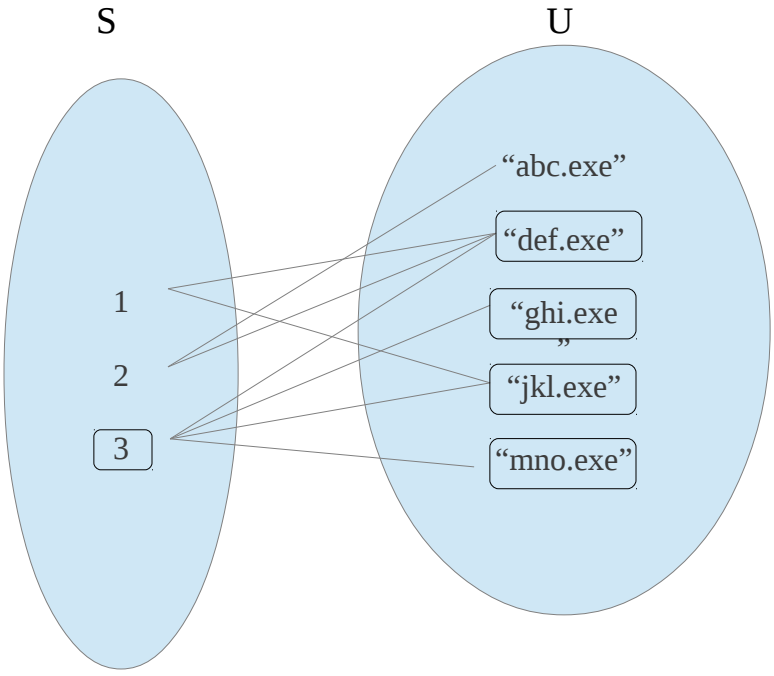
The above example can be diagrammatically shown as below.



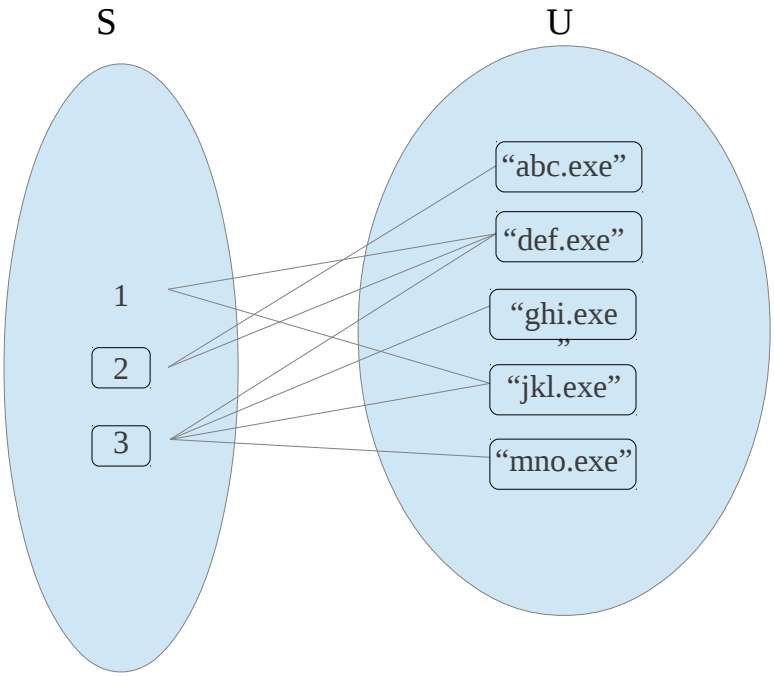
Let us imitate the way some of our brains might operate to solve this problem, the greedy algorithm. We first identify the largest subset and m
ark it with the 'covered' tag along with the elements of the universal set associated with the subset. Here, we mark S_3 and its elements in the universal set. We add S_3 to the distilled inputs set, I.

At this intermediate state, I = {S_3}. But we still have elements in the universal set that are yet to be covered.

Now we are in a state depicted by the following diagram. Boxes define the 'covered' state.



Now, of all the subsets that remain untagged, the subset that has the largest number of untagged elements is S_2. After tagging S_2 and adding it to I, we find that all the elements in the universal set have been covered.



Therefore the output of the above problem is

$I = \{S_2, S_3\}$

2.3 Complexity [3] :

A problem is said to be NP-Complete if it is in the set of NP problems and also in the set of NP-Hard problems. The abbreviation NP stands for Non-Deterministic Polynomial-time.

2.3.1 P [4]: The Complexity class of a decision problem that can be solved in polynomial time. That is, given a problem belonging to P class, the answer (say, yes or no) can be obtained in polynomial time.

Example : Eulerian cycle decision problem

Given a graph G, an Eulerian cycle is a cycle in G that passes through all the nodes (possibly more than once) and every edge of G exactly once. The decision problem deals with finding if such a cycle exists in a graph. The solution involves two steps. First we check if the graph is connected and then we check if each vertex has even degree. Both the steps occur in polynomial time and hence, the

decision problem belongs to class P.

2.3.2 NP [5] : NP is the set of all decision problems for which the instances where the answer is "yes" have proofs of the fact that the answer is indeed "yes", the proof being verifiable in polynomial time. This means that if someone gives us an instance of the problem and a certificate to the answer being yes, we can check that it is correct in polynomial time.

Example : Clique problem is NP

This problem of verifying whether a given graph G with n nodes has a clique of size k. Cliques are complete subgraphs in a graph. That is, given a graph and a statement that the graph contains a clique of size k, the statement can be verified in polynomial time.

2.3.3 NP-Complete [6] : NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time.

Example: Hamiltonian cycle problem

This is to determine whether a given graph has a Hamiltonian circuit (a closed path that traverses through each vertex only once)

2.3.4 NP-Hard [7] : A problem X is NP-Hard if there is an NP-Complete problem Y such that Y is reducible to X in polynomial time. They are more or less equal to or worser than NP problems. Even if someone suggests a solution to an NP-Hard problem it will still take more than polynomial time to verify if it is right.

Example : Traveling salesman problem

Finding the shortest tour through n cities with known positive integer distances between them.

2.4 Brute Force attack :

The simplest and most inefficient solution for set cover problem, as for any other problem, is by brute force attack. Brute force method is a straight forward approach which is directly based on the problem statement and the definitions of the concepts involved. This method iterates over all sets of the subsets to find the minimum set cover.

2.4.1 Integer Linear Programming [8]:

Integer Linear Programming problem is a problem of optimizing a linear function subject to linear inequality constraints constraints where the variables are restricted to be integers. The canonical form of an Integer Linear Programming can be expressed as

$$\text{maximize } \mathbf{c}^{\mathrm{T}}\mathbf{x}$$

```
            subject to Ax <= b,
                        x >= 0,
        and              x integer,
```

where the entries c, b and A are integers.

Example:

Maximize p = (1/2)x + 3y + z + 4w
subject to x + y + z + w <= 40
2x + y - z - w >= 10
w - y >= 10

The optimal solution to this problem is:

p = 115; x = 10, y = 10, z = 0, w = 20

2.4.2 Set cover problem defined in ILP

The set cover problem can be designed as a binary integer linear program where the unknown variables can take either a '0' or a '1'. In the set cover context, the unknown variables signify whether an input is an element in the “distilled” set or not. The objective would be to minimize the size of the “distilled” set. The constraints enforce the condition that all incorrect submissions that pass the public tests need to be covered.

Minimize p = sum(x[t] for t in inpset)
subject to
for i in fail_submissions:
sum(x[t] for t in fail_inputs[i])>= *1*

where, inpset is the set of all private inputs,
fail_submissions is the set of all incorrect submissions that pass the public tests,
fail_inputs[i] is the set of inputs that the submission i fails on.

Integer linear programming solutions are guaranteed to be optimal in cases where an optimal solution exists.

CHAPTER 3: GENERIC APPROACH

3.1 Introduction :

This chapter deals with the methods of how the project works in general. This covers the general way of how the method is followed, no matter whatever problem is under consideration. The central theme of the project falls in this chapter. At the end of this chapter, we will be able to understand the main objective and the working model of this project. As discussed in Chapter 1, the working structure can be mainly classified into four divisions. They are

- **Compiling the programs** : The source files are compiled and the corresponding executable ones are obtained.
- **Using diff** : In case of diff[9], we create a file with the aim that the outputs match their inputs.
- **Creating dictionary :** In this step, we create dictionaries by running the wrapper program,vinga on the collective inputs. This can be done using the Linux command, diff.
- **Applying set cover method** : The set cover approach is a mathematical model which is applied to receive a smaller set which is the goal of our project.

The steps that are to be taken before moving into the process are discussed in the following subsection and let us have short details before on the terminologies used.

Notes :

The .cc is a file extension for the source code for a program. The .exe is a file extension for an executable file format.

3.2 Preprocessing :

First, we should make sure that the directories should have the problem number in their name so that the identification would be easier. We have seen that there is a step called running through the wrapper program, vinga. So, in this preprocessing step, we have got to make the program compatible in order to make sure it works in a secured way. The wrapper program requires root level privileges so that it can enforce nobody level privileges on the programs submitted by the users. In linux environment, we first copy this wrapper program to the directory /usr/bin and change the permissions of the directory to root. We use few commands like

```
sudo cp vinga64 /usr/bin
sudo chown root:root /usr/bin/vinga64
sudo chmod +s /usr/bin/vinga64
```

Now, let us get into the function of the terms used above. sudo [10] is a Linux command used to execute the program as the root. One uses sudo when they need to run a command or program as root, but do not wish to log out or switch their entire shell to root privileges. cp [11] is the command used to copy the files to and directories to a particular destination directory. chown [12] is the command to changes the file owner and also the group ownership of the entire files in the path as in the command. chmod [13] command is used to set or modify the permission of the files. '+s' indicates that a user id is to be set to the files. The programs that are run in jutge.org[1] are treated as hazardous. They may cause threats to the environment in which they are run. Hence, they are run in a sandbox environment.

The “args” file:

For every problem, we create an “args” file for storing some preferences. These preferences represent the options that become relevant in different steps of execution.

For every problem,
"args" filename: Problem ID+"_args"

breakinp:

opt_inp (option for .inp files): default = 1

case 1 (based on string or character occurrences):
inp_char (char(s) to be used as delimiter for splitting) :: default = “\n”

case 2 (n followed by (c*n)+k lines):
inp_index (index, if any) :: default = 0
inp_c (c) :: default = 0
inp_k (k) :: default = 0

case 3 (n lines per input):
inp_n (n) :: default = 1

opt_cor (option for .cor files): default =1

case 1 (based on string or character occurrences):
cor_char (char(s) to be used as delimiter for splitting) :: default = “\n”

case 2 (n followed by (c*n)+k lines):
cor_index (index, if any) :: default = 0
cor_c (c) :: default = 1
cor_k (k) :: default = 0

case 3 (n lines per input):
cor_n (n) :: default = 1

createdict:

createdict_opt: default =2

case 1: based on split inputs and correct outputs .

case 2: based on diff .

setcover:

setcover_opt: default = 2

case 1: heuristics

case 2: pulp(ILP)

choice: “glpk”, “gurobi ” :: default = “glpk”

3.3Compilation :

Let us have a look on the algorithm for compilation to get an overview of what happens in this step.

```
Algorithm compile(pbm)
        //Compiles all the submitted source files of a problem
        //Input: A String, pbm, containing the problem ID
        //Output: Returns the name of the directory containing the compiled files

        root:="./" //Root must point to the directory which contains the problem directories(.sub
directories)
        for each sub_directory in root:
                if ".sub" and pbm in sub_directory_name:
                        dir_sub:=root+sub_directory_name+"/"
                        dir_cmp:=dir_sub.replace(".sub",".cmp")
                        if directory dir_cmp does not exist:
                                create directory dir_cmp
                        for file in dir_sub:
                                if ".cc" in file:
                                        file:=file.replace(".cc",".exe")
                                        outfile:=dir_cmp+file//filename of the executable file to be
created
                                        execute the command "g++ -D_JUDGE_ -DNDEBUG -O2 -o
"+outfile+" "+dir_sub+file
        return dir_cmp
```

The root must point to the directory which contains the problem directories (.sub directories). As given in the algorithm, for every subdirectory in the root, we check for the names. It should contain the problem name and also .sub in the directory name. In that case, we include it in the path and then the files are put in the dir_cmp directory where “.sub” is replaced with “.cmp” for every files. In case if dir_cmp does not exist, we create one and then put the files in the directory. For every file in the cmp_directory, we see that the extension “.cc” is replaced with “.exe”. Then, the file name for the executable file is created. We then execute the command as mentioned in the algorithm.

3.4 Using diff :

An alternative method of breaking inputs is by using diff command. The concept is having a same file for the inputs and to avoid the mismatch problems, we specify conditions such that the mismatch is prevented. For this, we use functions from the String matcher of Python Regular Expressions. The reference to this section is specified in the Bibliography.

Algorithm Breakinputs (pbm[, inp_opt[, cor_opt[, inp_char[, cor_char[, inp_c[, cor_c[, inp_k[, cor_k[, inp_index[, cor_index[, inp_n[, cor_n]]]]]]]]]])

//Break .inp and .cor files into independent inputs and correct outputs and store them separately

//**Input:** A String, pbm, containing the problem ID. Refer Section 4.2 for the optional arguments

//**Output:** Returns boolean 'True' at the end of execution.

// The .inp and the .cor files are split into files and stored as individual input files the names of which match with their correct output files.

```
root:="./"
dir_pbms:=[]
if inp_opt not passed:
        inp_opt:=1
if cor_opt not passed:
        cor_opt:=1
for every directory in root:
        if ".pbm" and pbm in dir:
                for every file in dir:
                        if ((".inp" in afile) or (".cor" in afile)) and not("buit" in afile):
                                content:=file.read()
                                dirname,fileext:=file.split(',')
                                count:=0 //count for creating sequential file-ids in each directory
                                cat:=fileext.strip('.') //Category variable to determine if the current file is ".inp" or ".cor"
                                if (cat=="inp"):
                                        opt:=inp_opt
                                elif (cat=="cor"):
                                        opt:=cor_opt
                                create directory dirname+"_inpcor" in dir
                                if(opt==1):
                                        if (cat=="inp"):
                                                if inp_char not passed:
                                                        inp_char:="\n"
                                                char:=inp_char
                                        elif (cat=="cor"):
                                                if cor_char not passed:
                                                        cor_char:="\n"
                                                char:=cor_char
                                        inplist:=content.split(char)
```

```
                    for each element in inplist:
                        create file count+fileext in dirname+"_inpcor"
directory
                        write element to file count+fileext
                        count:=count+1
                elif(opt==2):
                    pointer:=0
                    inplist:=content.split('\n')
                    if (cat=="inp"):
                        if inp_c not passed:
                            inp_c:=1
                        c:=inp_c
                        if inp_k not passed:
                            inp_k:=0
                        k:=inp_k
                    elif (cat=="cor"):
                        if cor_c not passed:
                            cor_c:=1
                        c:=cor_c
                        if cor_k not passed:
                            cor_k:=0
                        k:=cor_k
                    if(len(inplist[0].split())==1):
                        index:=0
                else:
                        if (cat=="inp"):
                            if inp_index not passed:
                                inp_index:=0
                            index:=inp_index
                        elif (cat=="cor"):
                            if cor_index not passed:
                                cor_index:=0
                            index:=cor_index
                    while(pointer<len(inplist)):
                        create file count+fileext in dirname+"_inpcor"
directory
                        if(index<len(inplist[pointer].split())):
                            end:=pointer+
(c*int((str(inplist[pointer]).split())[index]))+k+1
                        else:
                            break
                        if(end<len(inplist)):
                            for i in range(pointer,end):
                                writehandle.write(inplist[i]+"\n")
                        else:
                            break
                        count:=count+1
                        pointer:=pointer+
(c*int((str(inplist[pointer]).split())[index]))+k+1
                elif(opt==3):
                    pointer:=0
                    inplist:=content.split('\n')
                    if (cat=="inp"):
```

```
                    if inp_n not passed:
                        inp_n:=0
                    n:=inp_n
                elif (cat=="cor"):
                    if cor_n not passed:
                        cor_n:=0
                    n:=cor_n
                while(pointer<len(inplist)):
                    create file count+fileext in dirname+"_inpcor"
directory
                    if((pointer+n)<len(inplist)):
                        for i in range(pointer,pointer+n):
                            writehandle.write(inplist[i]+"\n")
                    else:
                        break
                    count:=count+1
                    pointer:=pointer+n
```

We enable the options for inp and the cor . For every file in the directory, we check if they are '.inp' file or '.cor' file. Then, we remove the '.' in the extension of the file name. And then, if the file is inp we take the option for inp and if it is cor we take the option for cor. Now, we check for the option.

1 - We see if it is inp or cor and check for the character. If nothing is specified by default, it is “\n” and then we break the inputs after every occurrence of the character specified. Then, separate files are created for each input with the name as given in the algorithm.

2 - We check for the k value and the index value. The k value specifies the break to be introduced at the k+1 th position and the index value indicates the value to be taken for the break to occur.

3 - This option mostly works for matrix operation. It checks for the value of n in a n*n matrix and then the break is made after every n lines.

A regular expression [14] (or RE) specifies a set of strings that matches it; the functions in this module let us check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing). The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form. For instance, let us have a glance on search attribute which we have used in our implementation.

re.search(string[, pos[, endpos]])

Scan through string looking for a location where this regular expression produces

a match, and return a corresponding 'MatchObject' instance. Return 'None' if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.
The optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the '^' pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.
The optional parameter endpos limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from pos to endpos - 1 will be searched for a match. If endpos is less than pos, no match will be found, otherwise, if rx is a compiled regular expression object, rx.search(string, 0, 50) is equivalent to rx.search(string[:50], 0).

In addition to the above method, we use functions like Popen and Pipe from the module 'subprocess' [15]. The subprocess module allows us to spawn new processes, connect to their input/output/error pipes , and obtain their return codes. This module intends to replace several other older modules and functions. Now, let us have a discussion on Popen and Pipe functions.

subprocess.Popen( ) :

The usage of 'Popen' would be in the form of "subprocess.Popen()". This executes a child program in a new process. We have used this in the following way.

```
    output = subprocess.Popen(cmd,
stdout=subprocess.PIPE ).communicate()[0]
```

In the arguments, stdout specifies the executed program's standard output. Valid values are 'PIPE' , an existing file descriptor (a positive integer), an exisitng file object and NONE. PIPE indicates that a new pipe to the child should be created. To have an idea of pipe, the description can be given as :

subprocess.PIPE :

Special value that can be used as the stdin, stdout, stderr argument to Popen and indicates that a pipe to the standard stream should be opened where stdin specifies the program's standard input, stdout specifies the program's standard output and stderr specifies the program's standard error stream.

Popen.communicate(input=NONE) :

Interact with process : Send data to stdin. Read data from stdout and stderr, until end of file is reached. Wait for the process to terminate. The optional input argument should be a string to be sent to the child process or 'NONE', if no data should be sent to the child.

communicate() returns a tuple (stdoutdata, stderrdata)

3.5 Creating dictionary :

In this step, we create dictionary by running through the wrapper program,vinga on the outputs received from the last step by two different methods.

Algorithm createDict (pbm[,dict_opt])

//Creates and returns a dictionary that associates the input with the set of file-ids of the compiled files that fail on each input

//**Input:** A String, pbm, containing the problem ID

//**Output:** Returns a dictionary that associates each input with the set of files that fail on that input

```
root="./"
errDict=dict()

if dict_opt not passed:
        dict-opt=2
if(dict_opt==1):
        for every directory in root:
                if ".cmp" and pbm in dir:
                                dir_cmp=root+dir+"/"
                                dir_pbm=dir_cmp.replace(".cmp",".pbm"))
                                sample_dir=dir_pbm+"sample_inpcor/"
                                for cmpfile in dir_cmp:
                                        for inpfile in sample_dir:
                                                run vinga with inpfile as the basename and the cmpfile as the command
                                                if not (execution == "ok"):
                                                        skip current cmpfile
                                        for every directory in root:
                                                if ("_inpcor" in directory name) and not ("sample_inpcor" == directory name):
                                                        for private_inpfile in directory:
                                                                run vinga with private_inpfile as the basename and the cmpfile as the command
                                                                if not (execution == "ok"):
                                                                        add cmpfile to errDict[private_inpfile]
                return errDict

elif(dict_opt==2):
        for every directory in root:
                if ".cmp" and pbm in dir:
```

```
                                        dir_cmp=root+dir+"/"
                                        dir_pbm=dir_cmp.replace(".cmp",".pbm"))
                                        sample_file=dir_pbm+"sample"
                                        for cmpfile in dir_cmp:
                                                run vinga with sample_file as the basename and the
cmpfile as the command
                                                run diff between the sample output file(sample_file.out)
and the sample correct output file(sample_file.cor)
                                                if (.out file and .cor file are similar):
                                                        for inpfile in dir_pbm:
                                                                if not "sample" in inpfile:
                                                                        run vinga with inpfile as the
basename and the cmpfile as the command
                                                                        run diff between the output
file(inpfile.out) and the correct output file(inpfile.cor)
                                                                        if not(.out file and .cor file are
similar):
                                                                                errlines=inpfile.read()
                                                                                for corfile_line_number in
diff output:
                                                                                        add cmpfile to
errDict[errlines[corfile_line_number-1]]
                        return errDict
```

Now, that the algorithm is shown above, let us have a simpler discussion on how it works. We have the set of compiled files as a result of previous steps. Now, the dictionary is created under two cases.

Case 1 :

This case is meant for the inputs that are obtained after breaking them. First, we check the sample_inpcor with each and every sample input. If they pass, we set sampleok to 1. After this step, we allow them to be tested with gran inputs. The gran_inpcor is now tested with each and every gran input. It is now important to check for the inputs that fail here. When an input fails, we add that particular fileid to the input key in the dictionary. In case, if the input key does not exist, we create one and initialize it to an empty set.

Case 2 :

In the second case, we perform the same step as the one in previous case but this time, it is done with the inputs that are obtained after performing the diff operation. We pass them through the sample and gran inputs for testing and do the same as in Case 1. The main operation of diff is to get the files that do not match. Once this is obtained, we add these fileid to the input key in the dictionary.

The main operation in this step is discussed and now, we shall see the information regrading vinga.

The wrapper program which allows us to validate the submissions against the correct outputs is named as 'vinga'. The general syntax of vinga can be given as follows:

```
vinga64 –basename = “input_package_basename” executable_file
```

In the above syntax, the basename refers to the input files like gran or sample. The output of this step would be the group of files with the names .out, .res, .log, .err. The out file is the output of the input file inp. The res file is the file with the information of the execution status, CPU time and clock time. The err file reports the list of errors, if any, during this step of execution. The log file contains a list of information regarding the memory, stack etc.

3.6Applying set cover method :

We have an overview of the set cover method from the second chapter. In this section we shall discuss the implementation of that method and the process that explains how the method is used and the expected outcome is obtained. The outcome of this step would be the reduced set of inputs which is the main goal of our project.

Now, let us move to the algorithm of the set cover.

```
Algorithm setcover (pbm,errDictfile[,setcover_opt[,inpChar[,choice]]])
        //returns the "distilled" set of inputs
        //Input:  A String, pbm, containing the problem ID,
        //        File, errDictfile, containing a dictionary of sets
        //Output: A subset of the keys of the dictionary
        root="./"
        if(setcover_opt not passed)
                setcover_opt = 2
        if(inp_char not passed)
                inp_char="\n"
        distilled=set()
        if (setcover_opt==1):
                read dictionary from file into a dictionary variable called errDict
                submissionset = union of errDict[i] for i in errDict.keys()
                while not((union of errDict[i] for i in distilled) == submissionset):
                        maxlen=0
                        for key in errDict:
                                if len(errDict[key])>maxlen:
                                        maxlen=len(errDict[key])
                                        point=key
                        outset=errDict.pop(point)
                        for key in errDict:
                                errDict[key]=errDict[key].difference(outset)
                        add point to distilled
                return distilled
```

```
elif (setcover_opt==2):
        read dictionary from file into a dictionary variable called errDict
        errDictGLPK=dict()
        for i in errDict.keys():
                for j in errDict[i]:
                        add i to errDictGLPK[j]
        inpset=errDict.keys()
        define binary linear program variable x [i] for i in inpset
        set ILP Objective to minimize sum(x[t] for t in inpset)
        for i in errDictGLPK.keys():
                add ILP Constraint that sum(x[t] for t in errDict[i]) > = 1
        if(choice not passed)
                choice = "glpk"
        if choice == "glpk"
                solve ILP in PuLP with GLPK as the solver
        elif choice == "gurobi"
                solve ILP in PuLP with Gurobi as the solver
        for t in inpset:
                if (x[t]==1):
                        add t to distilled
        return distilled
```

From the above algorithm, we see that we have assigned two options. If the option is 1, set cover method is done by heuristics and in the other case, we achieve this using Integer Linear Programming with the help of PuLP [16].

3.6.1 Integer Linear Programming (ILP):

In this method, we have used a modeler called PuLP and solvers like GLPK [17] and GUROBI [18] . The following section explains the usage in general and with reference to the implementation in our case study.

3.6.1.1 PuLP :

PuLP is an LP modeler written in Python. It can generate Mathematical Programming System(MPS) or Linear Programming (LP) files and call solvers to solve the program. The solvers can be GLPK, GUROBI, CPLEX, COIN-OR. By default, the solver that comes with the PuLP is COIN-OR.

3.6.2.2 Installation :

For details on this we refer to the following sites.
code.google.com and **www.coin-or.org**
The installation of PuLP is done by the following command.

```
$sudo easy_install -U PuLP
```

Following the above, we need to install all the PuLP functions into Python's callable modules. To do that, navigate to the respective directory and use the command

```
$sudo python setup.py install
```

Once PuLP has been installed, we can either use the COIN-OR solver that comes with the PuLP by default or install the solvers. In our case, we test the program with all the solvers in order to know the efficient one. Hence we install and try with them one by one. To test with the PuLP installation we type the following in the python interpreter.

```
>> import PuLP
 >> PuLP.PuLPTestAll()
```

This shows the list of solvers available for PuLP and testing for various solutions. The coding part in PuLP is discussed in this section. First, in order to proceed with the program we create new variables that are used. To create a new variable we use the function called PuLP.LpVariable() For example, if we want to create a variable 0<=X<=3, we put it in the following form.

```
x=PuLP.LpVariable('x', 0, 3)
```

And then, we create the problem with the function PuLP.LpProblem() For example, let us say we create a problem with the name 'Maths'. This is put in the below form.

```
prob=PuLP.LpProblem('Maths'.PuLP.LpMinimize)
```

LpMinimize is the constraint that has been added to this problem. In case, if we add an expression to the problem which is not a constraint, it becomes the objective of the problem. Then the problem is solved either with the default solver or we choose the solver to solve the problem. This is given by

```
status=prob.solve()
```

In case of choosing a solver for solving the problem, say GLPK, we mention it as

```
status=prob.solve(GLPK(msg=0))
```

In our project, we have used two different solvers apart from the default solver, COIN-OR. Let us have a discussion on the solvers used.

3.6.1.3 GLPK :

The GLPK (GNU Linear Programming Kit) package is intended for large-scale linear programming (LP), mixed integer programming (MIP) and other related problems. It is a set of routines written in ANSI C and organized in the form of a callable library. The reference is **en.wikibooks.org**

The installation of GLPK is done by using the command

```
$ sudo apt-get install glpk
```

If we just want the GLPSOL that is used to run the .lp(linear programming) file rather than downloading the GLPK package, we run the command below.

```
$ sudo apt-get install glpk-utils
```

In both the cases mentioned above, we confirm the GLPSOL by running the following.

```
$ glpsol --version
```

This gives the version number thus confirming the GLPSOL similar to the one below.

```
GLPSOL : GLPK LP/MIP  Solver 4.38
```

This can be done in another way. We download the .sig file available from the online resources and perform the following in order to make sure that the file that has been downloaded is intact.

```
gpg --verify glpk-4.32.tar.gz.sig
```

If that command fails because we do not have the public key, we run the following

command to import it.

gpg --keyserver keys.gnupg.net --recv-keys 5981E818

Once this is done, we rerun the previous command. In this way we set up the GLPK in our environment. Now, we can use PuLP and choose GLP solver to solve our integer linear program to achieve the reduced set cover.

3.6.1.4 Gurobi :

The Gurobi optimizer is a commercial optimization solver for Integer Linear Programming. The first step in installing the Gurobi Optimizer on a Linux system is to choose a destination directory. Once a destination directory has been chosen, the next step is to copy the Gurobi distribution to the destination directory and extract the contents. The details are obtained from **www.gurobi.com.**

Extraction is done with the following command:

```
tar xvfz gurobi5.5.0_linux64.tar.gz
```

This command will create a sub-directory gurobi550/linux64 that contains the complete Gurobi distribution. The file and directory names should of course be adjusted to reflect the actual distribution being installed (That is,, extracting the 32-bit Linux distribution (gurobi5.5.0_linux32.tar.gz) would create (/gurobi550/linux32).

The Gurobi Optimizer makes use of several executable files. In order to allow these files to be found when needed, you will have to modify a few environment variables:

- *GUROBI_HOME* should point to your <installdir>.
- *PATH* should be extended to include <installdir>/bin.
- *LD_LIBRARY_PATH* should be extended to include <installdir>/lib.

We, then execute the following commands in the terminal window.

```
export GUROBI_HOME = "/opt/gurobi550/linux64"
export PATH = "${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH = "${LD_LIBRARY_PATH}:$
{GUROBI_HOME}/lib"
```

The Gurobi Optimizer requires a license to run. We have several different license types. Our first step in setting up the license is to figure out what type we need. We can

use the free trial or academic license for the time being which can be created very easily. We create account in the Gurobi website and we are set to go. We visit either the Free Trial or the Free Academic tab on the Licenses page at our website. We first accept the license agreement, and then click on Request License. The new license will be visible immediately, under the Current tab. We can create as many trial or academic licenses as we like. Then, we are provided with the details of our License. To install the license in the computer where Gurobi optimizer is installed, we run the following command in the terminal.

```
grbgetkey key_number
```

For example, the license that we have obtained requires running the following command in the terminal.

```
grbgetkey ********-****-****-****-************
```

This can also be done with other type of solvers like Pyglpk and Yaposib. The details of the solvers and the installation process can be found form the links below.

YAPOSIB:

YAPOSIB is Yet Another Python OSI Binding. The details are obtained from https://code.google.com

PYGLPK :

PYGLPK is Python GLPK, yet another modified solver of GLPK. This can be learnt from tfinley.net

CHAPTER 4 : CASE STUDY – P18298

4.1 Introduction :

This chapter deals with the particular study. With reference to the previous chapter, we apply the methods discussed on a particular program and observe the results. We choose a particular program, say P18298 from Jutge.org and discuss the project from its point of view. This would help us understand the result obtained from this project and the work done in a better way.

Let us move to the problem that we have chosen for our case study. This problem is named as P18298 with the title “Roman numbers” in Jutge.org. The problem statement is to write a program that reads several numbers and prints their equivalent Roman number. It can be put in a more precise form with the input and the output samples.

Input :

Input consists of several numbers between 1 and 3999.

Output :

For each number, print its equivalent Roman number.

Sample Input	Sample Output
1 4 10 40 41 16 2708 999 3005	1 = I 4 = IV 10 = X 40 = XL 41 = XLI 16 = XVI 2708 = MMDCCVIII 999 = CMXCIX 3005 = MMMV

This problem comes in two languages, say the English and the Catalan where the main difference is the language of the statement and the data remains the same. Now, we discuss the various steps in the process that should be undergone which are generalized in the previous chapter.

We now move on to the working of this particular problem from our project point

of view.

First, let us have a general information regarding the problem P18298.

Problem ID : P18298

Languages : English, Catalan

No. of submissions in all languages : 883

The inputs and outputs are language-independent and hence only one set for all languages is necessary.

No. of sample inputs : 9

No. of gran inputs : 3999

4.2 Compilation :

Time taken : 7 minutes 52 seconds

4.3 Breaking Inputs/ Using diff :

inp_opt : 1
inp_char : “\n”
cor_opt : 1
cor_char : “\n”
Time taken : 5 seconds

4.4 Creating dictionary :

Case 1 :
dict_opt : 1
Time taken : 1 hour, 53 minutes and 15 seconds

Case 2 :
dict_opt : 2
Time taken : 1 minute 6 seconds

4.5 Applying set cover method :

Case 1 :

opt : 1
Using dictionary obtained from broken inputs:
Time taken to get the distilled inputs : 0 seconds
Number of distilled inputs : 11
Using dictionary obtained from diff command:
Time taken to get the distilled inputs : 1 second
Number of distilled inputs : 10

Case 2 :

opt : 2
Using dictionary obtained from broken inputs:
Time taken to get the distilled inputs : 5 minutes 22 seconds
GLPK time : 0.7 seconds
Number of distilled inputs : 9
Using dictionary obtained from diff command:
Time taken to get the distilled inputs : 5 minutes 22 seconds
GLPK time : 0.7 seconds
Number of distilled inputs : 7

CHAPTER 5 : CONCLUSION

5.1 Results and Comparison :

5.1.1 Break inputs and diff:

Break inputs apply to both single-line and multi-line layouts in inputs and outputs while diff command is applicable to single-line formats only. But, as clearly portrayed by the case studies, break inputs take a lot of time compared to the “diff” command. It is to be noted that breaking inputs as such is a one-time process for every test case. It need not be repeated once the test case has been included in the dictionary.

Break inputs also facilitates us to avert anomalies in the test case specification while diff can be misleaded. Therefore, break inputs comes with the guarantee on the resulting individual test cases and consequently on the error dictionary.

However, we recommend that break inputs can be done for problems involving complicated test cases while diff can be used for single-line formats.

5.1.2 Heuristics and Integer Linear Programming:

As always, when it comes to heuristics and integer linear programming, it is always a trade-off between time and optimality. Heuristics is fast while Integer linear programming is guaranteed to be optimal, that is, in cases where the optimal solution exists.

On comparing the two implementations for our set cover situation, we believe that the integer linear programming solution is better fearing the worst-case conditions where the solution provided by heuristics can turn out to be far from optimal and unrealistic.

5.2 Suggestions

The distilled set can be used as a private test case during verification of user programs as the programs are more likely to fail on these inputs. This will save us a lot of processing time that would have otherwise be spent on executing an incorrect program on the vast private test cases.

The authors who contribute problems to the jutge.org can be requested to avoid irregularities in the blueprint in the test cases. When working on large sets of user programs and test cases, spotting irregularities and eliminating them are tough.

References

[1] O. Giménez, J. Petit and S. Roura. (2006). Jutge::Documentation. Available: https://www.jutge.org/documentation/jutge.pdf.

[2] "Set Cover Problem – From Wikipedia, the free encyclopedia". http://en.wikipedia.org/wiki/Set_cover_problem.

[3] "Computational Complexity theory – From Wikipedia, the free encyclopedia". http://en.wikipedia.org/wiki/Computational_complexity_theory.

[4] "P Complexity – From Wikipedia, the free encyclopedia". http://en.wikipedia.org/wiki/P_%28complexity%29.

[5] "NP Complexity – From Wikipedia, the free encyclopedia".http://en.wikipedia.org/wiki/NP_%28complexity%29

[6] "NP Complete – From Wikipedia, the free encyclopedia".http://en.wikipedia.org/wiki/NP-complete

[7] "NP Hard – From Wikipedia, the free encyclopedia".http://en.wikipedia.org/wiki/NP-hard

[8] “Integer programming – From Wikipedia, the free encyclopedia”. http://en.wikipedia.org/wiki/Integer_programming

[9] "Linux/ Unix Command: diff”. http://linux.about.com/library/cmd/blcmdl1_diff.htm

[10] "Linux/ Unix Command: sudo”. http://linux.about.com/od/commands/l/blcmdl8_sudo.htm

[11] "Linux/ Unix Command: cp”. http://linux.about.com/od/commands/l/blcmdl1_cp.htm

[12] "Linux/ Unix Command: chown”.

http://linux.about.com/od/commands/l/blcmdl1_chown.htm

[13] "Linux/ Unix Command: chmod".
http://linux.about.com/od/commands/l/blcmdl1_chmod.htm

[14] "re – Regular expression operations".
http://docs.python.org/2/library/re.html

[15] "subprocess – Subprocess management".
http://docs.python.org/2/library/subprocess.html

[16] "Optimization with PuLP". http://pythonhosted.org/PuLP/

[17] "GLPK – GNU Linear Programming Kit".
http://www.gnu.org/software/glpk/glpk.html

[18] http://www.gurobi.com/