**Faculty of Mathematics and Information Science Warsaw University of Technology**



**Algorithms and Computability Project**

prepared by:  
Kuśmierczyk Aleksander  
Sławińska Martyna  
Żaba Kornel

*Version no. 1.0*

*Date: 19.10.2017*

**History of changes**

***Example:***

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Author** | **Description** | **Version** |
| 19.10.2017 | Kuśmierczyk Aleksander  Sławińska Martyna  Żaba Kornel | First version | 1.0 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. Table of Contents

Table of Contents 2

1 Problem definition 3

1.1 Description of a problem 3

1.2 What is a project 3

1.3 What is an expert 3

1.4 Description of a problem using mathematical notions 3

2 Solution in the form of a pseudo-code 3

# Problem definition

## Description of a problem

The problem is to assign a set of experts to a set of projects. The assignment must be done in a way that covers the most of the specialization fields required by each project.   
The expert can be assigned to only one project, and only one its specialization might be used for that project. The project can have many experts assigned to it.  
The aim is to minimize the number of unused experts or to minimize the number of unfinished projects, i.e. the number of projects that are still lacking some experts.

## What is a project

The project is a vector of values, which belong to natural numbers. The number of elements in this vector depends on the features number declared in the input file.   
Each element of the project vector represents a specialization field required by the project, and its value is the number of experts needed in that domain.

## What is an expert

The expert is a vector of Boolean values. The number of elements in this vector depends on the features number declared in the input file.  
Each element of the expert vector represents a specialization field, which belongs to the expert’s area of expertise. The value of 1 signifies that the expert specializes in a given feature, while 0 indicates the lack of knowledge in given field.

## Description of a problem using mathematical notions

u – number of projects  
v – number of experts  
w – number of features  
SP – set of pairs (Expert, Project) representing that the expert is working on the project

A pair (Ek,Pi) represents that the kth expert working in a ith project.

A pair (Ek,pi) represents that the kth expert working in a ith field of a project.

∀i∈N, i≤v Ei = [e1, e2, … , ew]-1 ∀j ej ∈ {0,1}

∀i∈N, i≤u Pi = [p1, p2, … , pw]-1 ∀j pj ∈ N

∀k∈N, k≤v ∀ i,j∈N, i≤u, j≤u, i≠j (Ek,Pi) ∈ SP ⇒ (Ek,Pj) ∉ SP

∀k∈N, k≤v ∀ i,j∈N, i≤w, j≤w, i≠j ∃l ∈N, l≤u pi ∈ Pl (Ek,pi) ⇒

~∃ j∈N, j≤w pj ∈ Pl (Ek,pj)

The solution to the problem is the result of a function, which minimizes the number of not assigned experts.

F(u,v,w,E,P) = min (#x : x∈E ∧  x∉SP)

# Solution in the form of a pseudo-code

E – set of expert vectors

P – set of project vectors

sum – vector of summed values of expert vectors

sum = sumExperts( E )

calculatingWeight( E ) // results are saved in the experts

sort( E ) // according to occurrence and weight

indProj = 0 // index of the project to which experts are being currently assigned

oldProjects = 0

projectSum = SumProjectVectors( P, featureCount ) // vector sum of projects

sumProjects = SumProjects( P ) // sum of all the features used in all the projects

usedExperts = null // vector that stores already used experts

while((AreExpertsAssignable(projectsSum, sumExperts, usedFeatures)) && (experts are not finished || projects are not finished)) // while loop runs until experts or projects are not finished and until an experts can be assigned

{

setUsedFeatures(projectSum, sumExperts, usedFeatures) // function marks already used feature

        oldProjects = sumProjects

        diff = 0

        indSmall = getSmallestIndex(sumExperts, usedFeatures)

        if( indSmall == -1 )

                break;

        if( P[ indProj ] requires feature with index indSmall )

        {

                    If( P[ indProj ][ indSmall ] < sumExperts[ indSmall ] )

                                diff = P[ indProj ][ indSmall ]

                                sumProjects[ indSmall ] -= P[ indProj ][ indSmall ]

                                P[ indProj ][ indSmall ] = 0

                        projectSum[ indSmall ] -= diff

                    else

                                diff = sumExperts[ indSmall ]

                                P[ indProj ][ indSmall ] -= sumExperts[ indSmall ]

                                sumProjects -= sumExperts[ indSmall ]

                        projectSum[ indSmall ] -= diff

        }

        removeExperts( E, usedExperts, diff, P[ indProj ], indSmall ) // function removes used

        experts from the set of experts E

removeExpertsFromSum( usedExperts, sumExperts, diff ) // function removes all the features of used experts from the sumExperts vector

        if( indProj == P.Count - 1 )

                indProj = 0

        else

                indProj++

}

SumExperts(E)  // function which calculates the collective sum of all features for all experts

{

For ( int i = 0; i < numberOfFeatures; i++ )

{

For (int j =0; j < numberOfExperts; j++)

{

expertsSum[i] += E[j][i] // E[j][i] is the jth expert and his ith feature

}

}

Return expertsSum; // we get vector with the sum of all the expert vectors.

}

sort(E) // this function sorts the experts according to their assigned weight and their features vector the comparison is made with the use of the following comparator and the implemented vector frequency[] holding the indexes of features sorted according to their frequency within the sumExperts(E) function

Compare (Expert1, Expert2)

{

For (int i=0; i < numberOfFeatures; i++)

{

If (Expert1[frequency[i]]== 1 && Expert2[frequency[i]] == 0)

Return 1; //we check if Expert1 has an ith feature, if yes and Expert 2 does not, he will be superior in comparison

Else if (Expert1[frequency[i]] == 0&& Expert2[frequency[i]] == 1)

Return -1; // reversed situation of the pervious comment

}

Return Expert1.weight - Expert2.weight; // finally we sort them according to weight.

}

CalculateWeights(E, int[] expertSum) // assigns weight to an expert

Weight of an expert is sum of experts features multiplied by their weight(the more rare the feature, the greater the weight)

{

sumTemp = expertSum.copy()

sumWeight = int[expertSum.length]

featureCount = total number of unique features

index = 0

while(index != featureCount)

{

//we assign weight value to each feature, the rarest feature has highest weight assigned, the most common feature has lowest weight

max = -1, maxIndex = -1

for(i =0; i < featureCount; i++)

{

if(sumTemp[i] > max)

{

max = sumTemp[i]

maxIndex = i

}

}

sumTemp[i] = -1

sumWeight[i] = index + 1

index++

}

for(j = 0; j < E.length; j++)

{

weight = 0

for(y = 0; y < expertSum.length; y++)

{

E[j].weight += E[j][y] \* sumWeight[y] //E[j][y] is value of experts feature(0 if expert does not have it, 1 if he has it)

}

}

}

bool IsAssignable(int[] sumProjects, int[] sumExperts) // this function checks if any projects still requires an expert and if there are any experts left

returns true if experts should be assigned, false if not

{

for(i = 0; i < sumProjects.length; i++)

{

if(sumProjects[i] > 0 and sumExperts[i] > 0)

{

return true

}

}

return false

}

setUsedFeatures(int[] sumExperts, int[] sumProjects, int[] usedFeatures)//

Sets given feature to 1, if it should be no longer assigned(no expert has it or no project requires it)

{

for(i = 0; i < sumProjects.length; i++)

{

if(sumProjects[i] > 0 and SumExperts[i] > 0)

{

usedFeatures[i] = 1

}

}

GetSmallestIndex( sumExperts, usedFeatures ) // this function returns the index of feature that is owned by smallest number of experts

{

        min = int.MaxValue

        index = -1

        for( i = 0; I < sumExperts.Count; i++ )

        {

if( (there are still some experts owning feature i) && (feature i is not marked as used in usedFeatures) && s(umExperts[i] < min) )

{

        min = sumExperts[i]

        index = i

}

        }

        return index

}

RemoveExperts( experts, usedExperts, amountToBeRemoved, assignedProject, featureUsed)

//this function removes the experts from the initial set and moves them to the usedExperts set which contains experts who are already assigned to a project with the use of specific feature.

{

countdown = amountToBeRemoved

j =0;

While ( countdown != 0)

If (experts[j][featureUsed] == 1) //we are determining the indexes of experts who have the designated feature and are the first ones to encounter in the experts set)

{

expertsToRemove.add(j);

countdown--;

}

for (int i = expertsToRemove.Count - 1; i >= 0; i--) // we remove the experts from the initial set and add them to the usedExperts set

           {

               usedExperts.Add(experts[expertsToRemove[i]])

               experts.RemoveAt(expertsToRemove[i]);

           }

for (int i = numberOfUsedExperts - 1; i > numberOfUsedExperts  - amountToBeRemoved - 1; i--)

//we assign the experts to the projects they were used for and specify the feature they were used with.

           {

               usedExperts[i].assignedProject = assignedProject;

               usedExperts[i].featureUsed = featureUsed;

           }

}

RemoveExpertsFromSum(E, int[] expertSum, int diff) // removes features of diff assigned experts from the sum in order to update available features in expert set

{

for(i = E.length - 1; i > E.length - diff - 1; i--)

{

for(j = 0; j < expertSum.length; j++)

{

expertSum[j] -= E[i][j]

}

}

}