Faculty of Mathematics and Information Science Warsaw University of Technology

Algorithms and Computability Project

prepared by:
Kuśmierczyk Aleksander
Sławińska Martyna
Żaba Kornel

Version no. 1.1

Date: 06.11.2017

History of changes

Date	Author	Description	Version
19.10.2017	Kuśmierczyk Aleksander Sławińska Martyna Żaba Kornel	First version	1.0
06.11.2017	Kuśmierczyk Aleksander Sławińska Martyna Żaba Kornel	Second version	1.1

Table of Contents

1 Problem definition

1.1 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.

1.2 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.

1.3 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.

1.4 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 (E_k,P_i) represents that the kth expert working in a ith project.

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

$$\forall_{i\in N,\, i\leq v}\, E_i = [e_1, e_2, \dots , e_w]^{-1} \quad \forall_j\, e_j \in \{0,1\}$$

$$\forall_{i\in N,\, i\le u}\ P_i = [p_1, p_2, \dots, p_w]^{-1} \quad \forall_j\ p_j \in N$$

$$\forall_{k\in N,\, k\le v}\ \forall_{i,j\in N,\, i\le u,\, j\le u,\, i\ne j}\ (E_k,P_i) \in SP \Rightarrow (E_k,P_j) \notin SP$$

$$\forall_{k\in N,\, k\le v}\ \forall_{i,j\in N,\, i\le w,\, j\le w,\, i\ne j}\ \exists_{l\in N,\, l\le u}\ p_i \in P_l\ \ (E_k,p_i) \Rightarrow$$

$$\sim\exists_{j\in N,\, j\le w}\ p_j \in P_l\ (E_k,p_j)$$

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

$F(u,v,w,E,P) = \min\ (\#x : x\in E \wedge x\notin SP)$

2 Solution in the form of a pseudo-code

```
Assignment - a class representing a tree structure consisting
of every possible expert Assignment. Each Assignment has an
expert with a specified project and feature he used in it
(except for the root node).

Run ( Projects, Experts, featureCount) // main function
{
        projectSum = SumProjectVectors( Projects, featureCount)//
        vector holding the sum of all projects vectors.
        RecursiveAssign(usedExperts, Experts, Assignment,
        featureCount)
        optimalPath = a.OptimalPath()
        usedExperts = FindUsedExperts(optimalPath)
        AssignExpertsToProjects(Projects, UsedExperts)
}

RecursiveAssign(expertsUsed, Experts, Assignment,
featureCount)) // usedExperts is the set of experts assigned
to the projects, a is an Assignment, the function generates
tree structure held in Assignment and representing any
possible combination of experts.
{
        expertsUsedCopy = expertsUsed;
        for(i=0; i<NumberOfExperts,i++)
        {
                if(!expertsUsed contains (Experts[i])
                        for(j=0; j<featureCount; j++)
                        {
                                A = Assignment.add (expert[i]),k)
                                if( a!=NULL)
```

```
                    {
                         expertsUsed.add(Experts[i])
                         RecursiveAssign(UsedExperts,Experts,A
                    ,featureCount)
                    }
               }
     }
}

FindUsedExperts(Assignment) // retrieving the experts from the
tree structure in the reversed order in which they were
assigned.
{
     while(Assignment != null)
     {
          expertsUsed.Add(Assignment.expert)
          Assignment=Assignment.parent;
     }
     Return expertsUsed;
}

AssignExpertsToProjects(Projects, expertsUsed)//function that
recreates the assignment of the experts to projects basing on
the feature they were used with and the sequence.
{
     for(i=expertsUsedCount - 1; i >= 0; i--)
     {
          for(j=0; j < ProjectsCount; j++)
          {
               if(Projects[j][ExpertsUsed[i].featureUsed>0)
               {
                    Projects[j][ExpertsUsed[i].featureUsed--;
                    UsedExperts[i].AssignedProject =
project[j];
                    break;
               }
          }
     }
}

Assignment.Add(Expert, indexFeature)//Adding additional
assignments to the parent node in order to fulfil the features
needed by the projects
{
     if(projectSum[indexFeature]>0)
     {
          Expert.featureUsed = indexFeature;
          Assignment = new Assignment(projectSum)
          Assignment.Expert = Expert;
          Assignment.projectSum[indexFeature]--;
          this.AssignmentList.Add(Assignment);
```

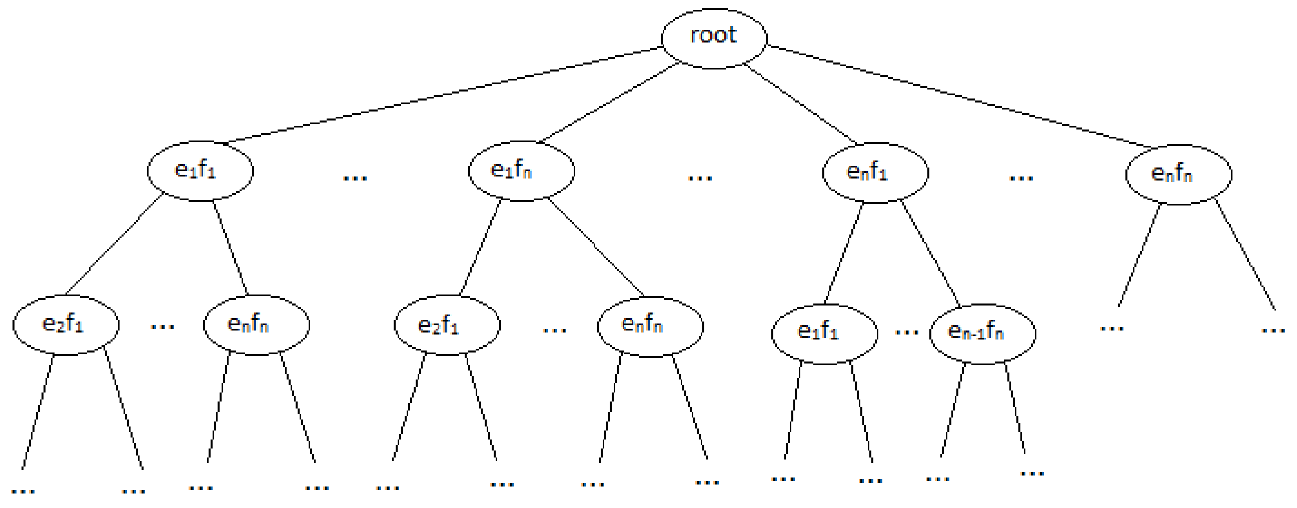
```
            Assignment.parent = this;
            Return Assignment;
        }
        Return Null;
}

Assignment.OptimalPath()// the initial call for the
RecursiveLongestPath function
{
        maxPathLength = 0
        pathLength = 0
        Assignment = RecursiveLongestPath(maxPathLength,
        pathLength, this);
        return Assignment ;

}

RecursiveLongestPath(maxPathLength, pathLength,
currentAssignemnt) // finds the longest path to be created
from the root node of an Assignment – finding the solution in
which the most experts are used.
{
        pathLength++;
        Assignment = null;
        if(current.AssignmentList.Count == 0)
        {
            if (pathLength > maxPathLength)
            {
                maxPathLength = pathLength;
                return current;
            }
        }
        foreach(child in Current.AssignmentList)
        {
            Assignment ret = RecursiveLongestPath(ref
            maxPathLength, pathLength, child);
            if (ret != null)
            Assignment= ret;
        }
        return Assignment;
}
```

3 Proof of correctness of an algorithm

Proof by contradiction:
The assumption is that our algorithm is not correct.

Definition of the correct algorithm:
Correct algorithm is the algorithm that chooses such combination of projects and experts' features that the number of unused experts is minimized.

Our algorithm creates the tree of assignments such that each path from the root up to the leaf is one combination of projects and experts' features.
The tree is created such that each node's children include every expert's features that are available (every expert = every expert that was not used before on that path). Thanks to this structure of the tree, we are able to find every possible combination of projects and experts' features.



e_if_j is the node of the tree only if expert i has a value 1 at the index j (i.e. experts i specializes in feature j)

Note:
The set of children of a node containing feature of an expert k does not include this expert's features and any other previously used expert's features.

After finding every possible combination of projects and experts' features (one combination = one path from the root up to leaf of the above tree), we choose the longest path from the root up to the leaf, which is equivalent to choosing the combination with the largest number of experts used. Aforesaid implies that we minimize the number of unused experts.

From above description, one may conclude that our algorithm is correct because it generates every possible combination for the given input and then finds the longest path (the maximum number of experts used). Thus it minimizes the number of unused experts.

4 Analysis of the complexity of an algorithm

Following is the calculation of time complexity of each function and total time of the algorithm.

- SumProjectVectors(Projects P, Features F)

  1. Function iterates through each project (P times)

  2. For each project function iterates through each feature (P * F times)

  Total: T(P, F) = O(P * F)

- Assignment.constructor(Features F)

  1. Function copies list of length F (F times)

  Total: T(F) = O(F)

- Assignment.addExpert(Expert E, int index)

  1. Function copies list of length F (F times)

  Total: T(F) = O(F)

- RecursiveAssign(Experts Eused, Experts E, Assignment A, Features F, int max, Assignment maxLeaf)

  1. Function copies list of length E (E times)

  2. Function iterates through list of length E (E times)

  3. Function searches list of length E (E * E times)

  4. Function iterates through list of length F (E * F times)

  5. Function adds Expert, by copying a list (E * E * F times)

  6. Function calls itself, for each expert and feature (E * F times) until it creates tree of height E

  Total: T(E,F) = O((E*E*F) * (E*F)^E)

- FindUsedExperts(Experts E)

  1. Function creates list of E experts (E times)

  Total: T(E) = O(E)

- AssignExpertsToProjects(Experts E, Projects P)

  1. Function iterates through each Expert (E times)

  2. For each expert, function iterates through each Project (P times)

  Total: $T(E, P) = O(E * P)$

Total complexity of an algorithm:

$T(E, P, F) = O((P * F) + F + F + ((E*E*F) * (E*F)^E) + E + (E*P))$