UNIVERSITE SIDI MOHAMED BEN ABDELLAH
FACULTÉ DES SCIENCES ET TECHNIQUES FÈS

DÉPARTEMENT D'INFORMATIQUE

PROJET DE FIN D'ETUDES
MASTER SCIENCES ET TECHNIQUES
SYSTÈMES INTELLIGENTS & RÉSEAUX

L'ÉVALUATION DE PROGRAMME C À TRAVERS LA MESURE DE SIMILARITÉ SÉMANTIQUE DANS LE CADRE DE L'ÉVALUATION AUTOMATIQUE DE PROGRAMMES BASÉE SUR LA MÉTHODE D'ANALYSE STATIQUE

LIEU DE STAGE : LABORATOIRE SYSTÈMES INTELLIGENTS ET APPLICATIONS L.S.I.A DE LA FACULTÉ DES SCIENCES ET TECHNIQUES DE FÈS

Réalisé par : Zainab OUFKIR && Adnane TAZI LABZOUR

SOUTENU LE : 24 JUIN 2015

ENCADRÉ PAR :

MR : BEN ABBOU RACHID

MR : ZAHI AZEDDINE

MME : MERNISSI ARIFI SARA

DEVANT LE JURY COMPOSÉ DE :

MR : BEN ABBOU RACHID

MR : ZAHI AZEDDINE

MME : LAMRINI LOUBNA

MME : CHAKER ILHAM

ANNÉE UNIVERSITAIRE 2014-2015

Résumé

L'analyse statique est une méthode qui permet d'examiner un programme sans passer par son exécution. Cette approche est utilisée dans le cadre de l'évaluation automatique de programmes.

Parmi les différentes techniques catégorisées dans le cadre de cette approche, on trouve l'analyse structurelle de programme, qui consiste à comparer le programme évalué avec d'autres programmes appartenant au plan solution en passant par la représentation par graphe.

L'objectif de ce projet et de mesurer la similarité sémantique dans le cadre de l'évaluation automatique de programmes (entre solution modèle et solution suggérée).

Une nouvelle approche a été proposée permettant de vérifier la similarité sémantique de programmes en s'appuyant sur leurs exécutions symboliques. Cette dernière consiste à vérifier les comportements des programmes et de les comparer sur le plan fonctionnel afin de relever les différences sémantiques et non textuelles.

Mots clés :

Similarité sémantique, Similarité structurelle, évaluation automatique, exécution symbolique, analyse statique, programmation C.

Abstract

Static analysis is a method that allows evaluate a program without executing it. This approach is used for the automatic grading of programs.

One of different techniques that are categorized in this approach, we find structural analysis of programs, that compares the evaluated programs belonging to the solution plan by using the graph representation.

The main target of this project is to measure the semantic similarity as part of the automatic grading of programs between the model solution and the suggested solution.

A new approach is proposed allowing the verification of programs semantic similarity that depends on their symbolic executions. This approach involves verification of the program's components and comparing them at a functional level in order to raise the semantics differences not the textual ones.

Key words:

Semantic similarity, structural similarity, automatic grading, symbolic execution, static analysis, programing C

Dédicace

On dédie ce travail à toute personne ayant contribué de près ou de loin soit par sa présence, son encouragement ou par son aide afin de nous donner le courage de réaliser ce travail et d'arriver jusqu'au bout, nos parents avec leur soutien tout au long de notre cursus universitaire, nos professeurs avec leur patience, conseils et persévérance depuis notre 1 ère année master sans oublier nos chers encadrant qui grâce à eux et à leur suivis nous avons pu réaliser une très grande partie de ce travail.

Ce modeste travail présente notre reconnaissance, nos remerciements et notre amour pour vous tous.

Remerciements

Nous remercions Dieu qui nous a donné le courage et la patience pour accomplir ce travail.

Nous tenons à remercier vivement les membres du jury qui ont accepté d'évaluer et de juger notre travail.

Par ailleurs, nous tenons à remercier nos encadrant M. BENABBOU Rachid, M. ZAHI Azeddine et Mme MERNISSI ARIFI Sara pour leurs efforts, leur présence et leur sens de du partage déployé en vue d'améliorer notre travail.

Sans oublier de remercier nos chers professeurs du département informatique pour leur accompagnement afin de nous garantir une formation de qualité dans les meilleures conditions et de nous mettre sur la bonne voie pour la suite de notre carrière.

Un grand merci à nos parents, amis, frères et sœurs. Ce travail est le fruit de votre amour, votre attention et votre soutien.

Enfin, nous remercions toute personne ayant contribué de près ou de loin à la réalisation de ce travail.

Table des matières

Tables des figures :

Liste des abréviations :

APAS : (Automated Programming Assessment Systems), système d'évaluation automatisé des programmes

CFG : graphe de flot de contrôle

SDG : graphe de dépendance de programme

PC : Path condition

Présentation du Laboratoire des Systèmes Intelligents et Applications.

Acronyme

Laboratoire **S**ystèmes **I**ntelligents & **A**pplications : **LSIA**

Responsable

Pr. Jamal KHARROUBI

Présentation

Le laboratoire SIA, crée en 2011, est une unité de Recherche du Centre d'Etudes Doctorales en Sciences et Techniques de l'Ingénieur domicilié à la Faculté des Sciences et Techniques de Fès et regroupant 17 laboratoires de recherche tous accrédités par l'Université Sidi Mohamed Ben Abdellah de Fès, et domiciliés à la Facultés des Sciences et Techniques, l'Ecole Supérieure de Technologie et la Faculté Poly-disciplinaire de Taza.
Le LSIA est composé de 13 enseignants-chercheurs du département d'Informatique de la FST de Fès et de 8 doctorants. Cette imbrication étroite entre enseignement et recherche, est un élément essentiel de la dynamique du laboratoire.
Les thématiques de recherche se situent au cœur des Sciences et Technologies de l'Information et de la Communication et s'articulent essentiellement autour des thématiques de recherche des enseignants chercheurs du laboratoire et assure une large couverture thématique présentant un atout très important pour le LSIA.

Equipes

Le laboratoire est composé de 3 équipes de recherche :

- Systèmes de Communication et Traitement de Connaissances (SCTC)
- environnement Intelligents & Applications (VIA)
- Vision Artificielle & Systèmes Embarqués (VASE)

Formation

Licence Sciences et Techniques « Génie Informatique »
Master Sciences et Techniques « Systèmes Intelligents & Réseaux »

Introduction

Ce travail a été effectué dans le cadre d'un projet de fin d'études au sein du laboratoire Systèmes Intelligents et Applications L.S.I.A de la Faculté des Sciences et Techniques de Fès.

Le projet a été suggéré dans le cadre d'une thèse de doctorat intitulée : l'évaluation automatique de programme, étude de cas de la programmation en langage C préparée par Mme. MERNISSI ARIFI Sara.

Le nombre de plus en plus croissant des étudiants inscrits dans les cours de programmation provoquant ainsi une quantité considérable de copies à corriger, pose un sérieux problème aux enseignants notamment lorsque la durée de correction est très limitée. Le passage à l'automatisation de la tâche de l'évaluation des programmes devient de plus en plus une nécessité.

Le recours à l'évaluation automatique de programme offre plusieurs avantages à savoir que la note attribuée par un système d'évaluation automatique peut être plus objective, mais aussi les feed-back fournis instantanément à l'étudiant présentent pour lui une aide précieuse. Pour cela, plusieurs recherches ont été effectuées dans ce cadre.

L'évaluation automatique s'appuie sur deux approches, l'analyse dynamique et l'analyse statique. Dans le cadre de notre travail de recherche, nous nous basons sur les méthodes d'analyse statique dans le but de concevoir et de mettre en œuvre un système d'évaluation automatique de programme écrits en langage C.

Dans ce sens, nous décrivons une nouvelle approche consistant à comparer deux programmes : le programme évalué (fourni par l'étudiant) et la solution modèle (fournie par l'évaluateur) en passant par leurs exécutions symboliques. L'exécution symbolique étant une méthode qui permet d'examiner statiquement un programme à travers la vérification des conditions et des données de ce dernier. Ce qui permet de relever l'équivalence des deux programmes d'un point de vue fonctionnel. L'enjeu de cette recherche réside dans la quantification du degré de similarité sémantique des deux programmes qui se traduit par une note.

Le calcul de la note a été calculé selon trois variantes que nous allons expliquer dans les chapitres suivants. Une comparaison entre les résultats des différentes formules de calcul de similarité a été dressée à la fin.

Nous commencerons notre travail par l'évaluation automatiques de programme C. Dans deuxième chapitre les méthodes d'analyse d'un programme seront abordées, le chapitre qui suive sera consacrée aux différentes approches d'analyse statique. Le quatrième chapitre sera réservé à la notion de la similarité sémantique entre les programmes .Le cinquième chapitre est le chapitre le plus important dans lequel notre approche adoptée sera décrite cette dernière s'appuie sur l'exécution symbolique et qui vérifie la sémantique d'un programme dans le dernier chapitre une application sera développée associée à l'approche adoptée.

Chapitre 1 : évaluation automatique des programmes

1. L'évaluation automatique des programmes

1.1 Introduction :

L'évaluation manuelle a toujours été une tâche banale mais qui nécessite une analyse rigoureuse. L'objectif de ce projet est d'évaluer un programme par un système automatique fiable qui donne une bonne approximation de la note en comparaison avec l'évaluation manuelle.

Un système d'évaluation automatisé des programmes (APAS, acronyme de : Automated Programming Assessment Systems) est un système assisté par ordinateur permettant de classer, d'évaluer et de vérifier l'exactitude des exercices de programmation des étudiants [1].

En effet, l'évaluation adoptée consiste à analyser le programme en s'appuyant sur une approche de l'analyse statique en comparant le programme de l'étudiant avec des solutions modèles. Le produit de cette comparaison étant une note reflétant le pourcentage de ressemblance entre les deux programmes.

1.2 Historique

L'évaluation automatique de programmes est une thématique traitée depuis plusieurs décennies. En 1960 Hollingsworth a publié le 1er rapport sur l'utilisation de « Automatic grader » dans « Rensselaer Polytechnical Institute » à New York. Quelques années plus tard d'autres systèmes ont vu le jour tel que BAGS.

En 1988, Ceilidh a vu le jour dans l'université de Nottingham. Ce dernier a été adopté par plus de 200 instituts d'enseignement supérieurs à travers 30 pays. Neuf ans plus tard, CourseMarker, un descendant de Ceilidh a été développé afin d'apporter une amélioration en termes de facilité d'utilisation et de mécanismes de feed-back à son prédécesseur.

Ultérieurement, d'autres systèmes d'évaluation automatique orientés web ont été implémentés tels que BOSS, Web-Cat plus robustes de point de vue sécurité, stratégie de feed-back et interopérabilité avec les LMS.

1.3 Avantages de l'utilisation des APAS :

Le recours à l'automatisation de l'évaluation des programmes présente des avantages énormes tels que :

- La réduction du temps consacré à l'évaluation des programmes, en effet la correction manuelle nécessite beaucoup de temps et souvent un travail fastidieux et banal.
- L'amélioration des compétences de l'étudiant en programmation à travers le renvoi immédiat de feed-back ce qui peut constituer un moyen d'auto-apprentissage pour ce dernier.
- Rendre la correction plus intègre et plus neutre, en effet l'évaluation manuelle peut être influencée par plusieurs facteurs tels que l'état d'esprit du correcteur, la fatigue, etc.

1.4 Synthèse

L'augmentation des copies à corriger dans les cours d'informatique nécessite trop d'effort et un temps considérable, aussi la correction manuelle reste toujours une tache banale.

La correction des copies dépend toujours de l'état d'esprit du correcteur ce qui amène parfois à des notations injustes du travail de l'étudiant.

L'objectif principale est d'avoir un système d'évaluation automatique de programme C qui donne une approximation proche de la notation manuelle des solutions des étudiants.

Chapitre 2 : Les méthodes d'analyse de programme

2. Les méthodes d'analyse de programme

Il existe deux approches d'analyse de programme, l'**analyse dynamique** nécessitant l'exécution du programme afin de vérifier son exactitude, contrairement à **l'analyse statique** permettant d'examiner un programme sans passer par son exécution.

2.1 Analyse dynamique

L'analyse dynamique implique l'exécution du code afin de vérifier son exactitude. Celle-ci est réalisée en utilisant des cas de tests différents et variées qui permettent une couverture maximale des chemins d'exécution de programmes.

C'est la méthode adoptée par la plupart des systèmes d'évaluation automatique de programmes tels que Ceilidh, TRY, BAGS, Kassandra.

On distingue deux approches différentes dans la méthode d'analyse dynamique:

- Le black-boxing qui consiste à considérer les sorties du programme dans sa totalité.
- Le grey-boxing qui consiste à valider les résultats de chaque fonction dans le programme et de générer le score final par rapport à ces résultats

2.1.1 Avantages de l'analyse dynamique

L'analyse dynamique est facile à implémenter, cela est démontré par son adoption par la plupart des systèmes d'évaluation automatique de programme.

Elle permet d'évaluer la performance du programme de point de vue des résultats générés comparés aux sorties attendues dans le cas de test.

2.1.2 Inconvénients de l'analyse dynamique

Les principaux inconvénients de l'analyse dynamique :

- Les risques relatifs à l'exécution en général du code source (l'exemple d'un dépassement au niveau du tampon ou débordement qui peuvent provoquer un arrêt ou une panne brusque d'un serveur et mettre les données en risque).
- L'handicap majeur de cette méthode : « si le programme ne se compile pas, alors il ne peut pas être évalué ».
- Le feed-back généré est limité aux sorties attendues du cas de test.
- On ne peut pas vérifier la conformité du programme quant aux exigences définies par l'évaluateur.

Chapitre 2 : Les méthodes d'analyse de programme

2.2 L'analyse statique

L'explosion du vol inaugural de la fusée Ariane 5 à cause d'un bug informatique a conduit à intégrer l'analyse statique dans le développement des logiciels critiques. En effet l'analyse statique permet de recueillir des informations sur le programme sans avoir à l'exécuter et par conséquent d'éliminer les risques liés à l'exécution de programme.

Parmi les outils qui s'appuient sur l'analyse statique [2] :

- **lint :** a été l'un des premiers outils d'analyse statique de code source. Sa création visait à remédier aux faiblesses des compilateurs des années 1970-1980.
- **AutoLEP** *:* outil s'appuie sur l'analyse dynamique et statique, Il peut donner des notes raisonnables à des programmes avec des erreurs syntaxiques ou logiques.

2.2.1 Avantages de l'analyse statique

Parmi les avantages de l'analyse statique, on cite :

- La prise en considération de toutes les exécutions possibles.
- La possibilité d'analyser le programme même si le code contient des erreurs contrairement à l'analyse dynamique.

2.2.2 Inconvénients de l'analyse statique

Parmi les inconvénients de l'analyse statique, on cite :

- La limite de l'application de l'analyse structurelle de programme du fait qu'il existe une variété de solutions pour le même problème.
- Complexe et difficile à appliquer dans le cadre de programmes complexes.

Chapitre 3 : Les Approches d'analyse statique

3. Les approches d'analyse statique

3.1 Les méthodes de l'analyse statique

Différentes méthodes sont distinguées dans le cadre de l'approche de l'analyse statique:

3.1.1 L'analyse de style

L'analyse de style: analyse la lisibilité du programme (les noms significatifs de variables, la présence de commentaires, les indentations, la portée des variables, etc.) afin que le programme soit compréhensible par d'autres utilisateurs.

Parmi les systèmes qui s'appuient sur l'analyse du style des programmes on trouve (Style++) pour le langage (C++) développé par (Ala-Mutka et. Al., 2004).

3.1.2 La détection d'erreurs

Les erreurs détectées à travers cette méthode sont les erreurs imperceptibles lors de la compilation, mais qui causent des problèmes lors de l'exécution (division par zéro, boucles infinies, etc.)

Ces erreurs ou ces bugs peuvent contribuer à un blocage ou un dysfonctionnement au niveau du système.

3.1.3 L'analyse des métriques

L'analyse des métriques : mesure les propriétés du programme afin de tester sa complexité et sa fiabilité (taille moyenne des instructions, fréquence de commentaires, nombre d'instructions dans une fonction, Nombre de classes, nombre des fonctions etc.).

3.1.4 L'analyse par mots-clés

L'analyse par mots-clés : examine le code source de manière à déterminer la présence des mots requis par l'évaluateur et qui doivent figurer dans le code source. (Exemple d'un programme dans lequel l'évaluateur exige l'utilisation de la structure Switch, et qui peut être écrit en utilisant un ensemble de if). Cette méthode donne la possibilité de vérifier si le programme respecte les exigences désignées par l'évaluateur.

3.1.5 L'analyse structurelle

L'analyse structurelle : mesure le degré de similarité en comparant le programme évalué avec d'autres programmes appartenant au plan solution, dit « experts » ou « modèles ».

Cette méthode en tant que méthode d'analyse statique, consiste à mesurer et à calculer le degré d'équivalence entre les programmes, en passant par la représentation graphique des programmes comparés.

3.2 Les formes de représentation de programmes

La représentation graphique de programme est une étape préliminaire à son analyse. Il existe différentes formes pour représenter et visualiser un programme :

3.2.1 Les Arbres syntaxiques

Un arbre syntaxique abstrait ou AST est un arbre dont les nœuds internes sont marqués par des opérateurs et dont les feuilles représentent les opérandes. Autrement dit, généralement, une feuille est une variable ou une constante.

Cette représentation est utilisée dans la recherche de similarité entre des programmes différents ou au sein du même programme. Après la création de l'AST, on cherche les sous arbres similaires à un arbre avec différents algorithmes. Enfin, on retourne le sous arbre commun entre les arbres comparés [13].

3.2.2 Les Graphes

Les graphes constituent un autre moyen pour la représentation d'un programme. A travers les graphes on peut visualiser la structure du programme afin de l'analyser et de le comparer avec d'autres programmes. Il existe plusieurs types de graphes:

3.2.2.1 Graphe de flot de contrôle

Le **graphe de flot de contrôle** (abrégé en GFC, Control Flow Graph ou CFG en anglais) est une représentation sous forme de graphe orienté, chaque nœud représente un bloc de base et chaque arc représente un flot de control.

NB :

Le graphe de flot de contrôle permet de préciser les chemins qui peuvent être empruntés dans un programme. Il est très utilisé dans les domaines d'optimisation de compilation et par certains outils d'analyse statique.

Exemple :

La figure 2 représente un graphe de flot de contrôle d'un programme qui demande à l'utilisateur d'entrer un nombre entier et affiche si le nombre est positif ou négatif.

```
int main (){
    int a;
    printf ("Entrer un nombre entier \n");
    scanf("%d",&a);

    if (a<=0)
        printf("le nombre que vous avez entré est négatif \n");

    else

        printf("le nombre que vous avez entré est positif \n");
    return 0;
}
```

Figure 1 : Programme permettant de déterminer si un nombre est positif ou négative

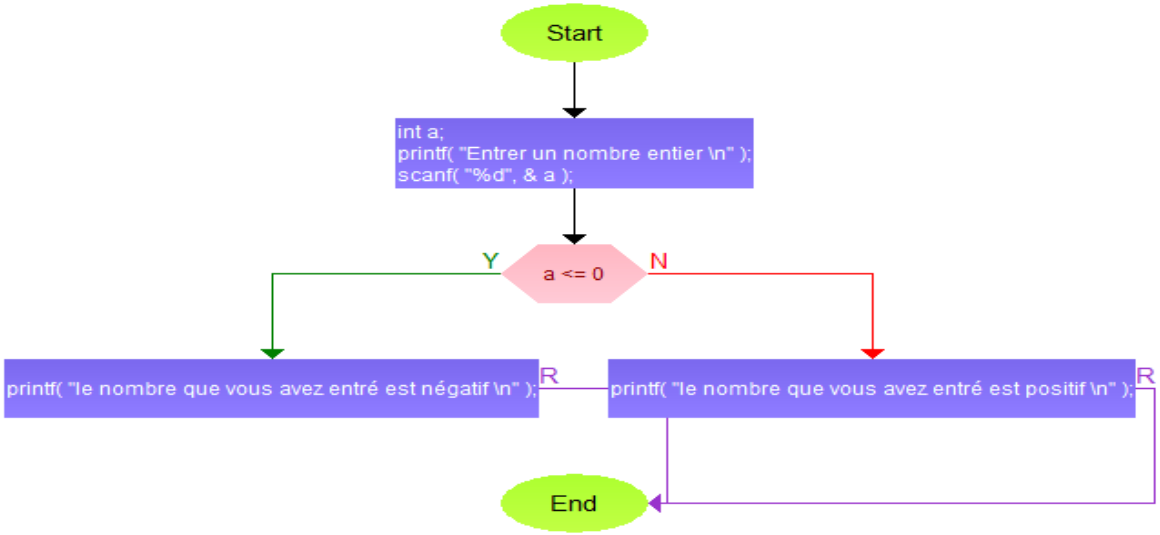


Figure 2 : Graphe de flot de contrôle du programme de la figure 1

3.2.2.2 Le graphe de dépendance de contrôle

Ce graphe montre quelles instructions seront exécutées en fonction de la valeur d'une expression dans le programme [3].

Les nœuds de ce graphe sont les mêmes que ceux du graphe de flot de contrôle.

Contrairement au graphe de flot de contrôle, le graphe de dépendance de control ne prend pas en compte le séquencement des instructions et le partitionnement sous forme de blocs de base.

Exemple de Graphe de dépendance de control

```
void main()
{
int x = 0;
int y = 1;
while (y < 5)
{
y = 2 * y;
x = x + 1;
}
printf ("%d",x);
printf ("%d",y);
}
```

Figure 3 : Programme qui calcul la valeur de x et y

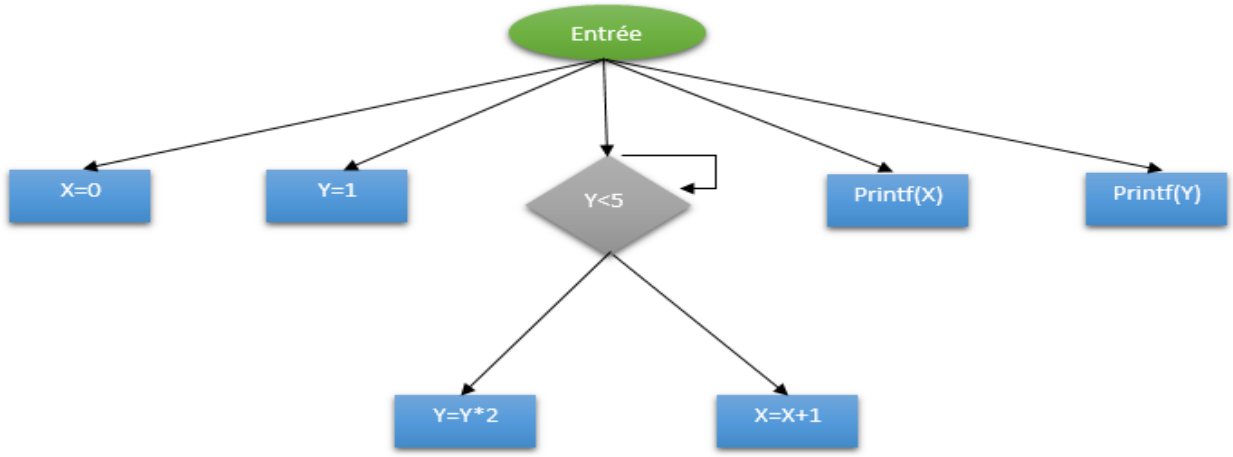


Figure 4 : Graphe de dépendance de contrôle associé au programme de la figure 3

3.2.2.3 Le graphe de dépendance de données

C'est un graphe dont les nœuds sont les mêmes que celui du graphe de flot de contrôle.

Dans ce graphe, un arc va de p vers q s'il est possible que la valeur d'une des variables modifiées à l'instruction p soit utilisée à l'instruction q sans qu'elle ne soit modifiée entre temps [3].

Le graphe de dépendance de données pour le même exercice présenté dans la figure 3 est le suivant:

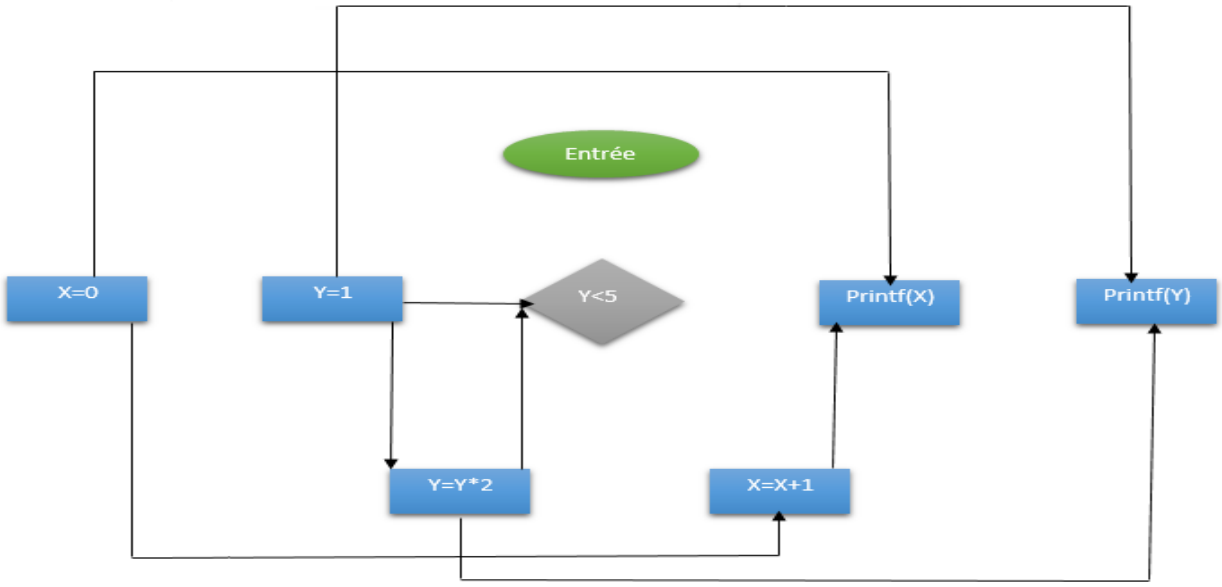


Figure 5 : Graphe de dépendance de donnée associé au programme figure 3

3.2.2.4 le graphe de dépendance du système de programme

C'est l'union du graphe de dépendance de contrôle et du graphe de dépendance de données

Ce type de graphe peut être utilisé pour représenter les programmes de procédure unique; un programme composé de la procédure main, sans autre procédure ou une fonction.

Exemple de graphe de dépendance de programme :

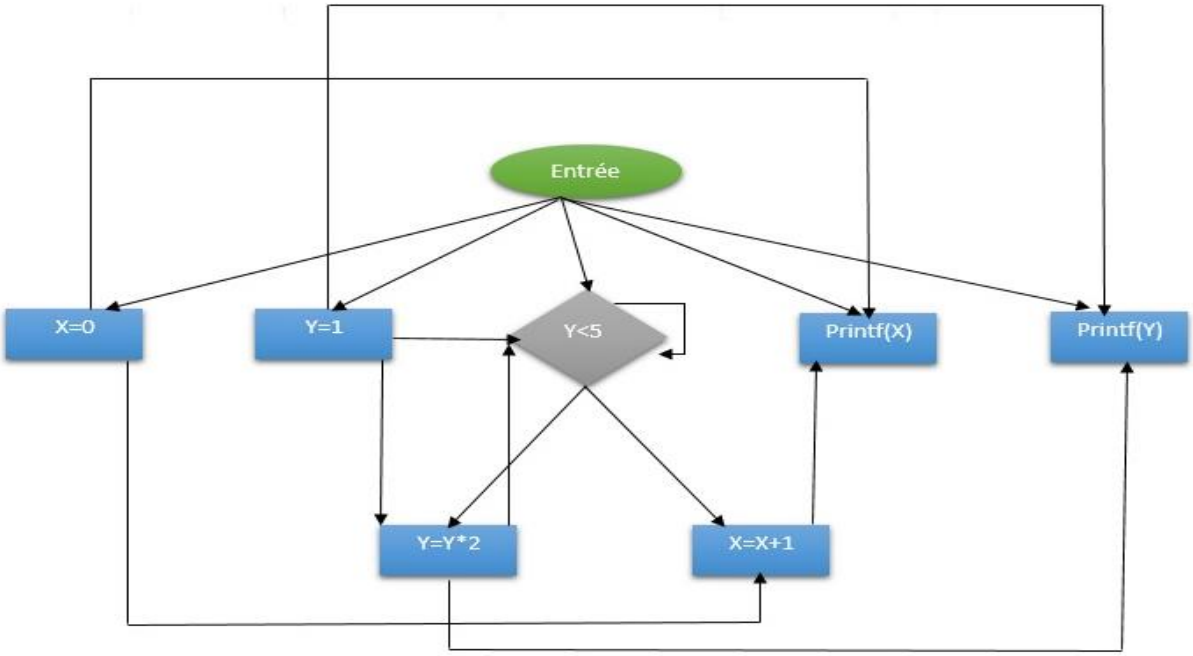


Figure 6: Graphe de dépendance du système de programme de la figure 3

Chapitre 4 : La notion de similarité

4. La notion de similarité

La détection de similarité consiste à comparer un programme solution proposé par un étudiant avec un ou plusieurs programmes proposés par l'évaluateur en tant que solutions modèles en s'appuyant sur les différentes approches de l'analyse statique.

La note présenterait une quantification du degré de similarité détectée entre le programme évalué et les programmes avec lesquels la comparaison a été faite. Différents types de similitudes peuvent être relevées entre les programmes : des similitudes sur le plan sémantique ou structurel.

4.1 La similarité structurelle

4.1.1 Introduction à la similarité structurelle

Plusieurs recherches ont été menées dans le cadre de la détection et du calcul de similarité entre programmes [5] [6] [7]. Un nombre important d'articles de recherches considèrent que la similarité dépend de la structure d'un programme et cela consiste à comparer les représentions structurelles (Graphes) des programmes des étudiants et les solutions des professeurs.

L'Approche consiste à :

- Construire un graphe (graphe de flot de control, graphe de dépendance de système) associé à un programme
- Réduire la diversité parasite (normalisation) exemple (supprimer des parties de codes redondantes).
- Comparer la représentation graphique du programme avec les représentations graphiques des solutions
- Attribuer une note à la comparaison.

4.1.2 Exemple Similarité structurelle :

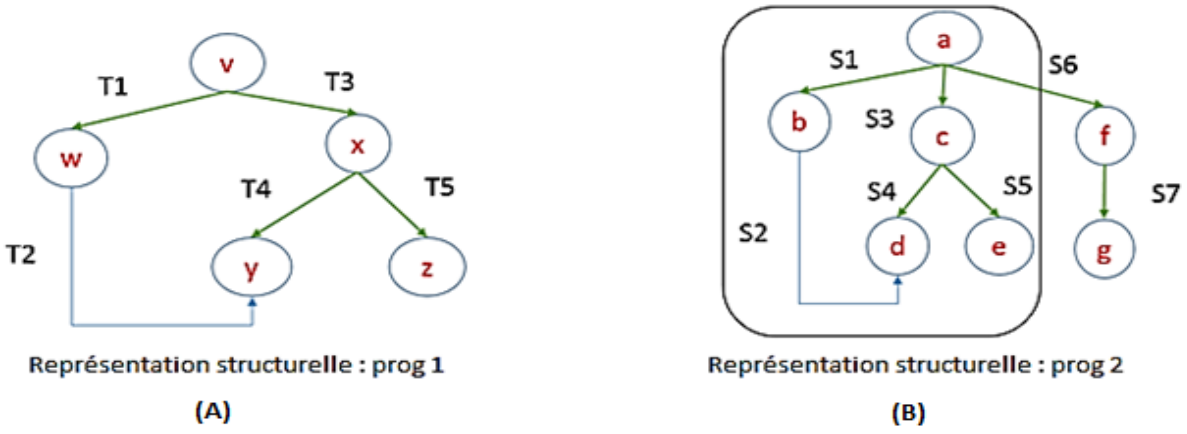


Figure 7: Similarité structurelle entre les graphes de deux programmes

Dans l'exemple on constate une ressemblance structurelle entre la représentation structurelle du programme1 (A) et une partie la représentation Structurelle de programme2 (B) (encadré) ce qui permet de relever une ressemblance structurelle partielle et par conséquent la note relative à cette ressemblance.

4.1.3 Les méthodes de mesure de similarité structurelle

Les algorithmes disponibles pour mesurer la similarité structurelle peuvent être divisés en deux [6]:

- Les méthodes traditionnelles basées sur l'équivalence exacte des sous-graphes (equivalence of subgraphs)
- Les méthodes modernes basées sur les heuristiques floues (fuzzy heuristics)

4.1.3.1 Les méthodes de similarité structurelles traditionnelles

Les méthodes traditionnelles examinent les mesures de la similarité structurelle reposant sur l'équivalence exacte de sous-graphes. On distingue différentes techniques dans ce cadre :

- Edit Distance Similarity : calcul métrique de la distance entre les nœuds des graphes.
- Maximum Common Isomorphic Subgraph : Le sous graphe commun maximal entre deux graphes.
- Tag and Path-Oriented Similarity:similarité basée sur la comparaison des documents XML des graphes, des balises XML représentant les nœuds et les arcs et les propriétés des graphes.

4.1.3.2 Les Méthodes de similarité structurelle modernes :

Les méthodes de similarité modernes reposent sur la mesure de similarité entre deux graphes, en comparant les matrices d'adjacences associées à ces derniers. Ces méthodes s'appuient sur la notion de scoring (pourcentage de ressemblance).

Différentes techniques sont considérées dans ce cadre:

- The Product Graph : deux graphes sont similaires si leurs voisins sont similaires.
- Weighted Assignment Similarity : est basé sur la recherche du voisin idéal entre chaque paire de sommets considérés en utilisant des scores.

4.1.4 Problématique de la mesure de similarité structurelle

On peut avoir deux programmes avec une représentation graphique identique mais qui n'ont pas le même comportement et ne donnent pas le même résultat.

Exemple :

```
max = 0;                          max = a[0];
for(i=0; i<n; i++)                for(i=1; i<n; i++)
   if(a[i] > max)                    if(a[i] > max)
      max = a[i];                       max = a[i];
```

(A) (B)

Figure 8: Exemple de deux programmes structurellement similaires

Deux programmes peuvent avoir une représentation graphique identique (par CFG ou SDG) malgré qu'ils soient différents sur le plan fonctionnel, c.-à-d. qu'ils ne se comportent pas de la même manière, et donc ne produisent pas les mêmes résultats.

En effet, dans la figure(8)(A)Le max est initialisé par 0, il se peut que le tableau contient des entiers négatifs contrairement à la figure(8)(B) qui commence par le 1er élément du tableau ce qui permet de dire que le programme(8)(A) n'est pas fonctionnel ,d'où la similarité structurelle ne vérifie pas l'équivalence des deux programmes sur le plan fonctionnel [4].

La similarité structurelle ne permet pas de vérifier le contenu des nœuds d'un graphe car elle s'intéresse plus à la structure du graphe. Par conséquent, cette méthode ne peut être adoptée dans le cadre de ce travail à travers lequel on cherche à mesurer la similarité de deux programmes sur le plan fonctionnel. Cela doit se faire à travers la comparaison des comportements au sein des nœuds.

4.2 La similarité sémantique

La similarité sémantique est une notion qui consiste à vérifier le contenu d'un programme en s'intéressant à la logique des instructions, des données et des conditions du programme.

Deux expressions peuvent être sémantiquement équivalentes même si elles sont écrites de manières différentes [11].

La similarité sémantique permettant de comparer un programme avec un ou plusieurs programmes modèles afin de s'assurer de leur similitude en se basant sur leurs sémantiques.

La similarité sémantique ne s'intéresse pas au contenu textuel du programme mais elle s'intéresse plus au comportement des instructions du programme.

La similarité sémantique est un nombre compris entre (0 et 1) [7] :

-Si le nombre égal à 1: les programmes sont sémantiquement équivalents.
-Si le nombre égal à 0: les programmes sont complètement différents.

Dans le cas de programmes complexes, on a besoin d'un grand nombre de modèles de solutions.

Chapitre 4 : La notion de similarité

Chaque développeur a ses propres habitudes de programmation et de nommage, il est difficile de déterminer la similarité par une simple comparaison du texte car deux programmes écrits différemment, peuvent être sémantiquement similaires, c.-à-d. Ils peuvent fournir les mêmes résultats. C'est la raison pour laquelle l'analyse sémantique présente un grand intérêt.

Le calcul de la similarité sémantique entre un programme de l'élève et un programme modèle est une tâche difficile en raison des variations syntaxiques. Pour résoudre ce problème, des transformations préservant la sémantique peuvent être effectuées sur les deux programmes.

Chapitre 5 : l'exécution symbolique

5. L'exécution symbolique

5.1 Définition de l'exécution symbolique:

C'est une approche qui s'appuie sur la sémantique et qui consiste à construire une représentation finie de tous les calculs d'un programme (transformations subies par les données ou par les conditions).

L'exécution symbolique [8] [9] [10] a pour but d'analyser statiquement un programme pour trouver des bugs ou prouver certaines propriétés du programme, de traiter les variables liées aux expressions symboliques (au lieu des valeurs concrètes), cela permet d'unifier les variables pour les comparer avec les mêmes variables des autres programmes.

L'exécution symbolique permet de:

- ✓ Enregistrer les conditions liant les instructions entre elles.
- ✓ Enregistrer aussi les variables symboliques associés aux variables existantes dans un programme.
- ✓ Parcourir tous les chemins d'exécution possibles d'un programme informatique.

5.2 Les notions de base de l'exécution symbolique

L'exécution symbolique contient deux notions essentielles :

- Le store symboliqueδs : contient les variables symboliques attribuées aux variables réelles, à chaque changement aux niveaux d'une variable le store change aussi.
  Exemple : supposons qu'on a trois variables a, b, c alors on obtient le store suivant :δs = {a=a_0 , b=b_0, c=c_0}
  Supposons qu'on a, c=a-b/2*b alors le store change et devient :
  δs = {a=a_0 , b=b_0, c=a_0-b_0/2*b_0}

- Le Path condition PC : contient l'historique des conditions collectées le long du chemin parcouru pour arriver à une instruction d'un programme.
  Exemple:
  Considérons l'instruction suivante:
  if(a>b && c>a){ } else {}
  PC_0= true PC est toujours initialisé par true ou par l'élément vide
  PC_1 =true && a_0> b_0 && c_0>a_0

5.2.1 Comment construire le Store?

a. On remplace les variables réelles par des variables symboliques
b. A chaque nouvelle instruction ou changement au niveau des données, le store change en mettant à jour la valeur de la variable symbolique concernée dans le store.
c. La mise à jour du store n'est pas effectuée lorsqu'il y a aucun changement au niveau des données du programme.

5.2.2 Comment construire le Path-Condition ?

a. On initialise le PC_0 par défauts avec TRUE
b. Chaque condition se trouvant dans une structure conditionnelle ou itérative, est ajoutée au PC après avoir remplacé chaque variable dans la condition par son symbole.
c. On cumule les conditions successives jusqu'à ce que tous les chemins possibles dans le programme soient parcourus.

L'exécution symbolique d'un programme est effectuée à travers sa représentation par le CFG composé d'un ensemble de blocs de base.

5.3 Décomposition d'un programme sous forme de bloc de base

Bloc de base : c'est une portion du programme, contenant un seul tenant sans saut ou cible de saut.

a. Les cibles de sauts marquent le début de blocs de base
b. Les sauts marquent la fin des blocs de bases

NB : le tenant représente une structure itérative ou conditionnelle et le saut représente un goto.

- Exemple décomposition en bloc de base

- 0: (A) Scanf("%d",t0);
- 1: (A) if (t0 mod 2 == 0){
- 2: (B) printf(" Paire");
- 3: (B) goto 5
- 4: (C) printf("Impaire");
- 5: (D) end program

La décomposition sous forme de bloc de base d'un programme est réalisée comme suite :

- **Bloc de Base (A)** : 0 et 1
- **Bloc de Base (B)** : 2 et 3
- **Bloc de Base (C)** : 4
- **Bloc de Base (D)** : 5

Pourquoi le graphe de flot de contrôle dans l'exécution symbolique?

❖ Pour préciser tous les chemins empruntés par le programme, cela permet de préciser la succession des blocs de base.

❖ Pour découper le programme sous forme de blocs de base, ces derniers sont utiles pour l'exécution symbolique (affecter les données (store) ainsi que les conditions (PC) associées à chaque bloc de base).

5.4 Exemple d'exécution symbolique d'un programme

Soit un programme qui permet de calculer le maximum de trois nombres.

Chapitre 5 :l'exécution symbolique

```c
#include<stdio.h>
#include<stdlib.h>
main(){
int a,b,c,max;
if(a>b)
max=a;
else
max=b;
if (c>max)
max=c;
printf("le max est %d ",max);
}
```

Figure 9 : Programme d'un exercice permettant de calculer le maximum de trois nombres

L'exécution symbolique associée et la suivante (les stores et les path condition sont indiqués sur le CFG représentant le programme) :

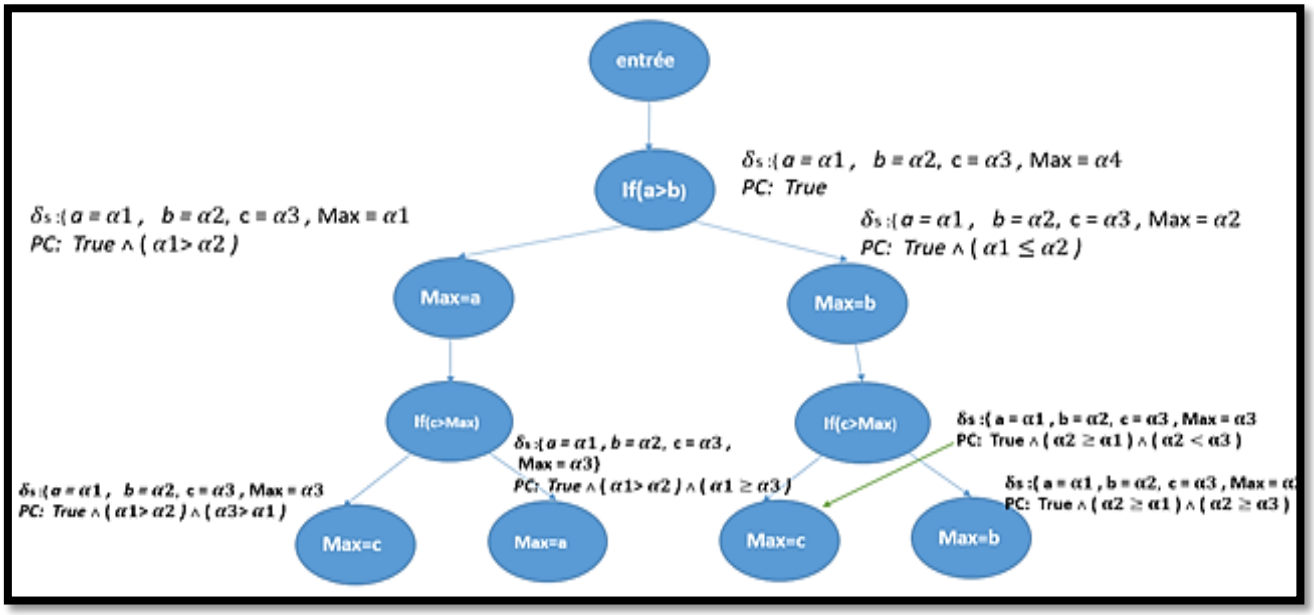


Figure 10: Graphe de flot de contrôle associé au programme qui calcule le max de 3 nombres

Lors de la génération des PCs , on s'est retrouvé avec deux blocs de base ayant le même store et amenant au même résultat avec deux PC différents, d'où la nécessité d'une normalisation afin d'aboutir à un PC unique. C'est une étape préliminaire qui précède le calcul de taux de ressemblance entre les blocs de base.

Dans l'exemple de la figure 10 (max =$\alpha 3$) on a :

- PC: True ∧ ($\alpha 1 > \alpha 2$) ∧ ($\alpha 3 > \alpha 1$) et PC: True ∧ ($\alpha 2 \geq \alpha 1$) ∧ ($\alpha 2 < \alpha 3$)

Afin d'éliminer les conditions redondantes et inutiles il nécessaire de faire une normalisation.

- PC: True ∧ ($\alpha 1 > \alpha 2$) ∧ ($\alpha 3 > \alpha 1$) ➔ $\alpha 3 > \alpha 2$

- PC: True ∧ ($\alpha 2 \geq \alpha 1$) ∧ ($\alpha 2 < \alpha 3$) ➔$\alpha 3 > \alpha 1$

Après une normalisation le résultat est le suivant :

- PC: True∧ $\alpha 3 > \alpha 2 \wedge \alpha 3 > \alpha 1$

5.5 La similarité sémantique à travers l'exécution symbolique

5.5.1 Introduction à la similarité sémantique à travers l'exécution symbolique

Notre approche s'appuie sur le graphe de flot de control pour explorer tous les chemins possibles dans un programme.

Le GFC découpe le programme sous forme de blocs de base où chaque nœud de graphe représente un bloc de base et le passage d'un nœud vers à un autre est lié à une condition.

Pour comparer deux programmes on essaye de comparer les blocs de base d'un programme proposé comme solution par le correcteur avec les blocs de base de programme généré par l'étudiant.

Le concept est le suivant :

Les Stores et les Path-conditions des deux programmes sont comparés deux à deux.

Les stores représentent les changements de données lors du passage d'un bloc de base à un autre, et le path-condition représente les conditions nécessaires pour passer d'un bloc de base à un autre.

Par exemple : si deux conditions sont réalisées parmi les 4 conditions, alors un pourcentage de2/4 *100 soit 50% est attribué comme taux de ressemblance. (Le taux de ressemblance étant entre 0% et 100%).

Lors de la comparaison le taux de ressemblance entre les blocs de base, on garde le taux le plus élevé jusqu'à ce qu'un pourcentage de ressemblance maximal soit trouvé.

La somme des pourcentages est calculée et divisé par le nombre des blocs de base maximal.

Pour avoir une note/20, le pourcentage est multiplié par 20 est divisé par 100.

5.5.2 Les variantes de calcules de taux de ressemblance

Concernant le calcul du taux de ressemblance représentant la similarité entre deux programmes, trois variantes ont été envisagées.

☒ La variante A

La 1 ère variante (variante A) consiste à attribuer :

- 50% de la note pour le Store
- 50% de la note pour le PC

Cette variante consiste à donner un pourcentage de 50% pour le store contenant les données et 50% pour le PC ou les conditions de passage à un bloc de base.

Le taux de ressemblance entre deux blocs de base se calcule de la manière suivante :

Soit :

T : Taux de ressemblance.

S : Pourcentage de ressemblance entre deux stores de deux blocs de base.

P : Pourcentage de ressemblance entre deux PC de deux blocs de base.

La formule de calcul de similarité adoptée dans le cadre de la variante A est comme suit :

$$T = (S + P)$$

☒ La variante Z

La 2éme variante (variante Z) consiste à :

- Exiger une ressemblance totale des stores
- Donner 100% au PC

Cette variante exige d'avoir une similarité complète entre les stores de deux blocs de base avant de comparer le PC ou les conditions de ces deux blocs. En d'autres termes, les PC de deux blocs de base sont comparés si et seulement si les stores associés à ces blocs sont entièrement équivalents.

Le taux de ressemblance entre deux blocs de base se calcule de la manière suivante :

Soit :

T : Taux de ressemblance.

S_{Bi} : Pourcentage de ressemblance du store associé bloc de base B_i.

S_{Bj} : Pourcentage de ressemblance du store associé bloc de base B_j.

P : Pourcentage de ressemblance entre deux PC de deux blocs de base.

La formule de calcul de similarité adoptée dans le cadre de la variante Z est comme suit :

$$T = P \text{ Si et seulement si } S_{Bi}=S_{Bj}$$

☒ La notion de pondération

Une 3éme méthode a été proposée et qui consiste à ajouter une technique de pondération aux méthodes présentées précédemment. Cela revient à adopter l'une des méthodes précédentes en attribuant un poids à chaque bloc de base.

Soit :

Poids (i) : poids associé à un bloc de base

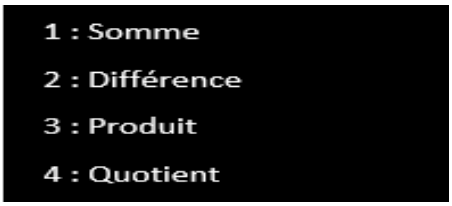
Pourcentage(i) : pourcentage de ressemblance entre deux blocs de base

$$T=\sum_{i=1}^{n}(poids(i) * pourcentage(i)) /100))$$

5.5.3 Exemple 1 : Deux programmes sémantiquement similaires

Deux programmes sémantiquement pareils impliquent que le taux de ressemblance soit égal à 100%.

Un programme qui demande à l'utilisateur d'entrer deux nombres réels puis lui propose le menu suivant :



Le programme ou la solution proposée par le correcteur est la suivante :

```
int main(){
     int x;
     float a,b;
     printf("Entrez le premier nombre\n");
     scanf("%f",&a);
     printf("Entrez le deuxième nombre\n");
     scanf("%f",&b);
     printf("choisir le numéro de l'opération:\n");
     printf("1:Somme \n");
     printf("2:Différence \n");
     printf("3:Produit \n");
     printf("4:Quotient \n");");
     G:scanf("%d",&x);
     switch(x){
          case 1:printf("la somme= %f",a+b); break;
          case 2:printf("la différence= %f",a-b);break;
          case 3:printf("le produit= %f",a*b); break;
          case 4:if (b==0) printf("division par 0!");
          else printf("le quotient= %f",a/b); break;
          default: printf("choix invalide");goto G;
     }
}
```

Figure 11 : Programme proposé par un professeur associé à l'exemple 1

Afin d'effectuer l'execution symbolique du programme proposé par le professeur, on suit les deux étapes suivantes:

La 1 ère etapeconsiste à :

- Visualiser leprogramme sous forme de GFC puis le découper en blocs de base.
- Identifier les blocs de base (A1,B1,C1, ect).

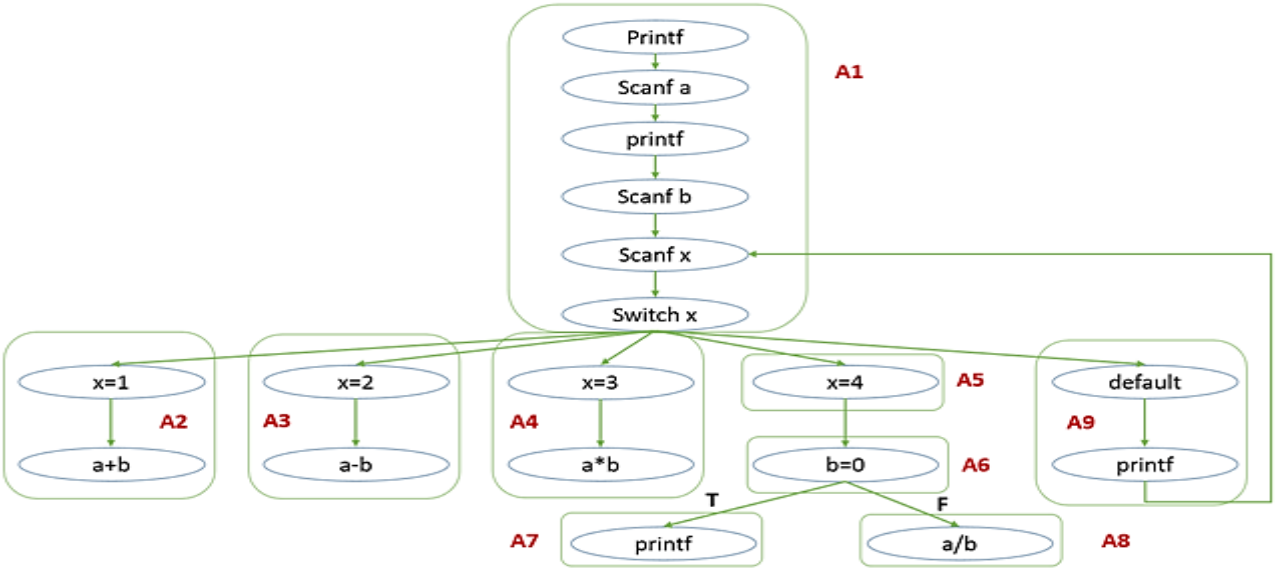


Figure 12 : Graphe de flot de contrôle du professeur de l'exemple 1

La 2éme étape consiste à :

- Faire une représentation symbolique des blocs de bases, c'est-à-dire construire les Stores et les PC .

A1 $\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4\}$

A2 *PC:* $\alpha4 = 1$
$\delta_s : \{\alpha1, \alpha2, \alpha3 = \alpha1 + \alpha2, \alpha4\}$

A3 *PC:* $\alpha4 = 2$
$\delta_s : \{\alpha1, \alpha2, \alpha3 = \alpha1 - \alpha2, \alpha4\}$

A4 *PC:* $\alpha4 = 3$
$\delta_s : \{\alpha1, \alpha2, \alpha3 = \alpha1 * \alpha2, \alpha4\}$

A5 *PC:* $\alpha4 = 4$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4\}$

A6 *PC:* $(\alpha4 = 1) \wedge (\alpha2 = 0)$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4\}$

A7 *PC:* $(\alpha4 = 1) \wedge (\alpha2 = 0)$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4\}$

A8 *PC:* $(\alpha4 = 1) \wedge (\alpha2 \; != 0)$
$\delta_s : \{\alpha1, \alpha2, \alpha3 = \alpha1/\alpha2, \alpha4\}$

A9 *PC:* $(\alpha4 \; != 1) \wedge (\alpha4 \; != 2) \wedge (\alpha4 \; != 3) \wedge (\alpha4 \; != 4)$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = \alpha5\}$

Figure 13 : L'exécution symbolique de la solution de l'exemple 1 proposée par le professeur

Chaque variable est initialisée par sa valeur symbolique cela a pour but d'unifier les variables afin de les comparer et de faire la correspondance entre les variables similaires portant des noms différents.

Le PC est remplie a partir des conditions existantes dans le programme. Le PC associé à chaque bloc de base exprime les conditions de passage d'un bloc de base a un autre .

Chapitre 5 :l'exécution symbolique

Soit le programme suivant proposé par un étudiant. On procéde de la même manière en répétant les deux étapes citées précedemment, concernant la représentation par graphe et l'exécution symbolique du programme.

```
int main(){
      int x;
      float a,b;
      printf("Entrez le premier nombre\n");
      scanf("%f",&a);
      printf("Entrez le deuxième nombre\n");
      scanf("%f",&b);
      printf("choisir le numéro de l'opération:\n");
      printf("1:Somme \n");
      printf("2:Différence \n");
      printf("3:Produit \n");
      printf("4:Quotient \n");");
      G:scanf("%d",&x);

            if(x==1)  printf("la somme= %f",a+b);

            if(x==2)  printf("la différence= %f",a-b);

            if(x==3)  printf("le produit= %f",a*b);

            if(x==4)  {
                        if (b==0) printf("division par 0!");
                        else printf("le quotient= %f",a/b);
                        }
            if(x!=1 && x!=2 && x!=3 && x!=4 ){

                        printf("choix invalide");goto G;

                                                }
            }
```

Figure 14 : Programme proposé par un étudiant associé à l'exemple 1

1ère étape : Le graphe de flot de contrôle de la solution proposée par un étudiant est le suivant :

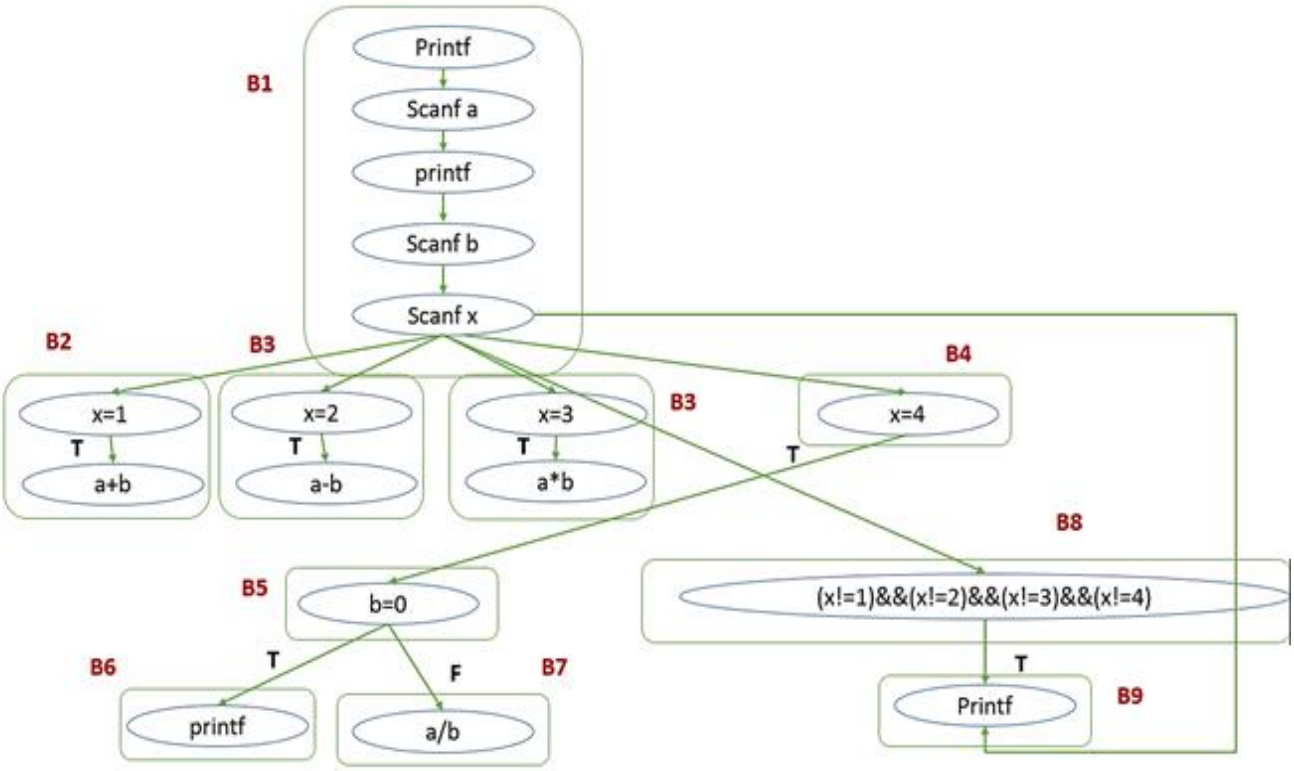


Figure 15: Graphe de flot de contrôle de la solution d'un étudiant de l'exemple 1

Chapitre 5 :l'exécution symbolique

2ème étape : La représentation symbolique des blocs de bases :

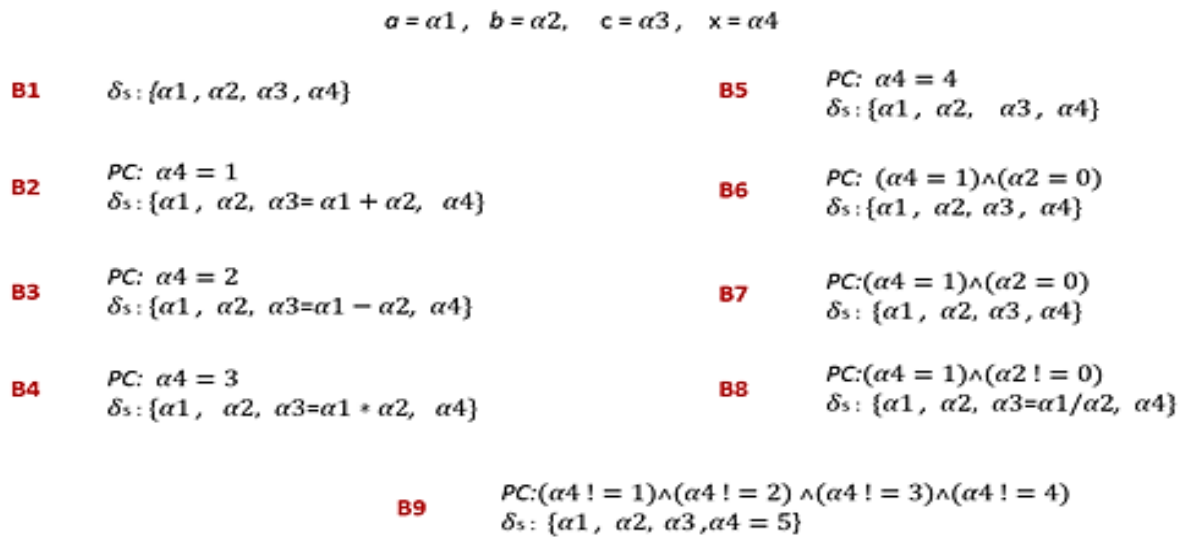


Figure 16 : l'exécution symbolique de la solution proposée par l'étudiant pour l'exemple 1

La similarité sémantique à travers l'execution symbolique :

Après avoir effectué l'exécution symbolique des blocs de base appartenant au programme du professeur et ceux constituant le programme proposé par l'étudiant, on procède à la comparaison séquentielle de tous les blocs de base en vue de detecter et de calculer la similarité sémantique entre chaque deux blocs de base appartenant aux deux programmes comparés.

La figure 17(A) représente les blocs de base de la solution du correcteur (proffesseurs) tandis que la figure 17(B) représente les blocs de base du programme proposé par un étudiant .

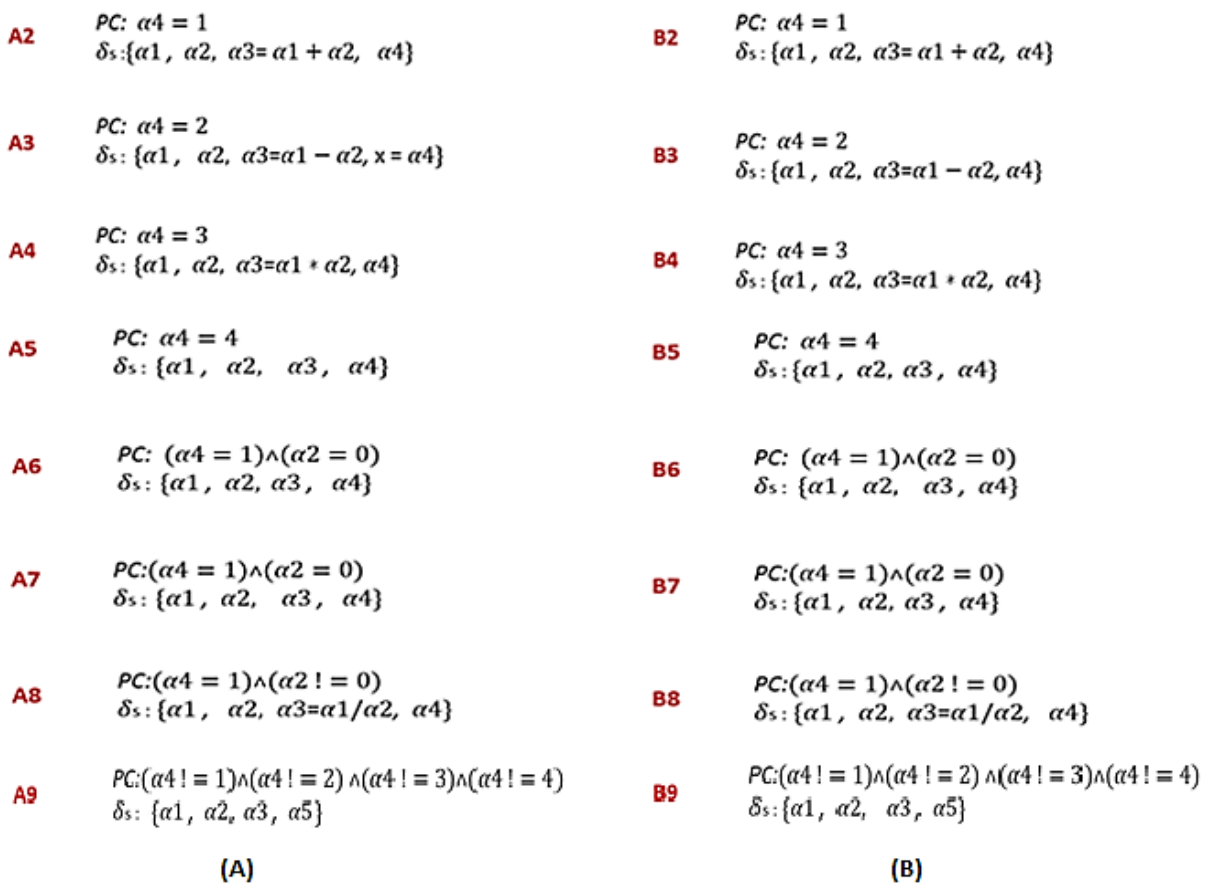


Figure 17 : l'exécution symbolique de deux solutions (professeurs et étudiant) de l'exemple 1

Chaque bloc de base appartenant à la figure (A) a son équivalent dans la figure (B). Cette équivalence est tributaire de la conformité du store et du PC des deux blocs de base comparés et appartenant chacun à un programme.

A travers cet exemple, on peut conclure que les deux programmes comparés sont textuellement différents, mais sémantiquement équivalents puisqu'ils ont les mêmes représentations symboliques .

5.5.4 Exemple 2 : l'exécution symbolique dans le cas de boucle :

Soit l'exercice suivant : « Ecrire un programme qui lit 5 notes comprises entre 0 et 20 et indique combien d'entre elles sont supérieures à la moyenne 10 ».

La solution modèle proposée par le professeur est la suivante :

Chapitre 5 :l'exécution symbolique

```
int main(){
    int a, i, c;
    c=0;
    for (i=0;i<5;i++){
        printf("entrer une note entre 0 20 \n");
        G:scanf("%d",&a);
        if(a>=0 && a<=20) {2
            if(a>=10) c++;
            }
        else {
        printf("note invalide! taper une note entre 0 et 20 \n");
        //scanf("%d",&a);
        goto G;
        }
    }
    printf("le nombre de notes supérieures à 10= %d",c);

return 0;
}
```

Figure 18 : Programme proposé par un professeur associé l'exemple 2

1ère étape : Le Graphe de flot de control de la solution du professeur:

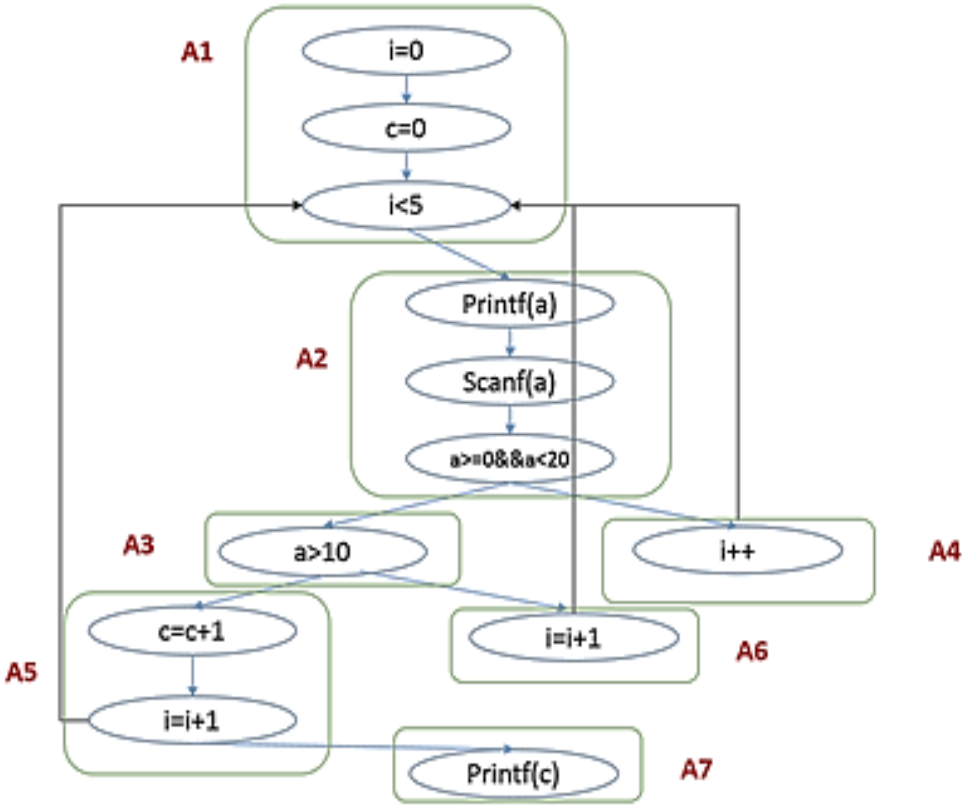


Figure 19 : Graphe de flot de contrôle du programme proposé professeur de l'exemple 2

2ème étape : la représentation symbolique du programme proposé professeur.

L'exécution symbolique associée à chaque bloc de base est représentée de la manière suivante:

A1 $\delta s : \{c=\alpha 1 = 0 \ , i = \alpha 2 = 0 \}$
PC: True

A2 $\delta s : \{c=\alpha 1 \ , i = \alpha 2, a = \alpha 3\}$
PC: $\alpha 2 < 5$

A3 $\delta s : \{c=\alpha 1 \ , i = \alpha 2, a = \alpha 3\}$
PC: $\alpha 2 < 5 \wedge \alpha 3 >= 0 \wedge \alpha 3 <= 20$

A4 $\delta s : \{c=\alpha 1 \ , i = \alpha 5, a = \alpha 3\}$
PC: $\alpha 2 < 5 \wedge (\alpha 3 < 0 \ || \ \alpha 3 > 20)$

A5 $\delta s : \{c=\alpha 4 i \ , i = \alpha 5, a = \alpha 3\}$
PC: $\alpha 2 < 5 \wedge \alpha 3 =< 0 \wedge \alpha 3 <= 20 \wedge \alpha 3 \geq 10$

A6 $\delta s : \{c=\alpha 1 \ , i = \alpha 5 , a = \alpha 3\}$
PC: $\alpha 2 < 5 \wedge \alpha 3 =< 0 \wedge \alpha 3 <= 20 \wedge \alpha 3 < 10$

A7 $\delta s : \{c= \sum_{i=1}^{5} \alpha 4 i , a = \alpha 3\}$
PC: $\alpha 5 >= 5$

Figure 20 : L'exécution symbolique de la solution de l'exemple 2 proposé par un professeur

Une solution proposée par un étudiant est représentée dans la figure 21

```
main()
{
      float a;
      int i,j=0;
      for(i=0;i<5;i++)
      {
                        printf("entrez une note entre 0 et 20 \n");
                        scanf("%f",&a);
                        if(a>10)
                        j++;
                        }
      printf("le nombre de note superieur a 10 est %d",j);
      system("pause");
      }
```

Figure 21 : Programme proposé par un étudiant associé à l'exemple 2

1ère étape : Le graphe de flot de contrôle de la solution est le suivant :

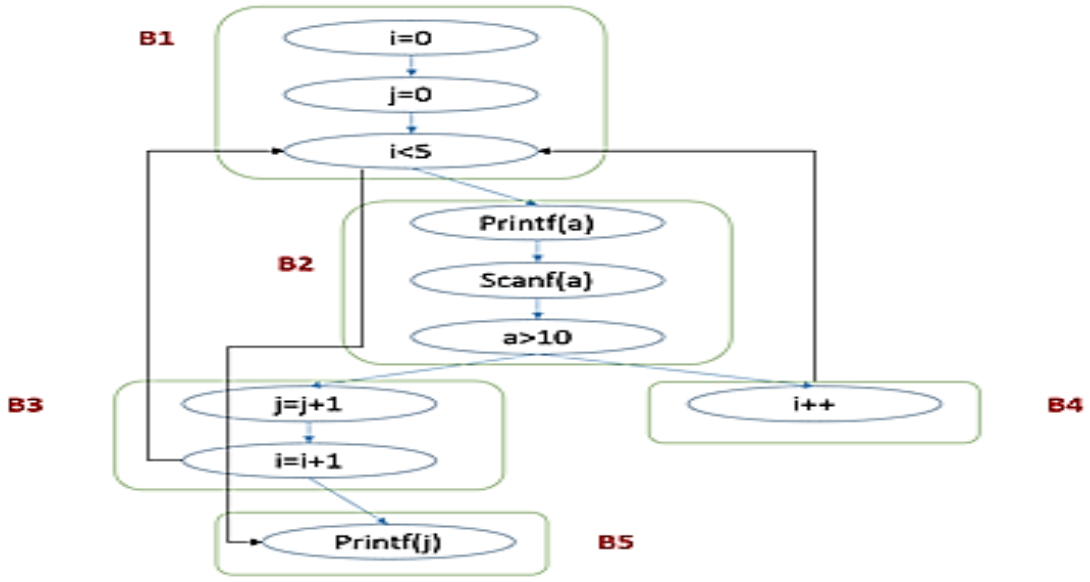


Figure 22: Graphe de flot de contrôle du programme étudiant de l'exemple 2

2ème étape : L'exécution symbolique associée à chaque bloc de base est représentée de la manière suivante :

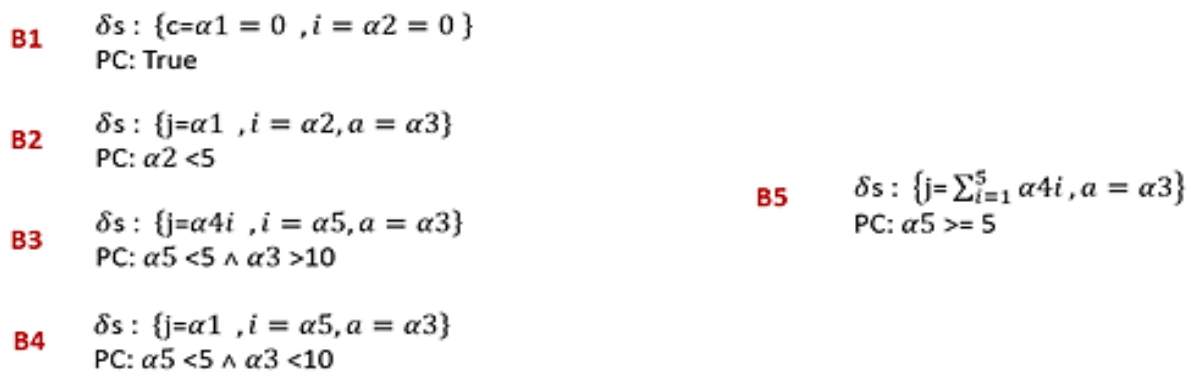


Figure 23 : l'exécution symbolique de la solution proposée par un étudiant pour l'exemple 2

Afin de mesurer l'exactitude et l'efficacité des variantes proposées dans le paragraphe 5.2.2 pour le calcul de similarité sémantique entre deux programmes, nous procéderons à l'expérimentation de chacune d'entre elles.

✓ Calcul de similarité en utilisant la 1 ère variante (variante A):

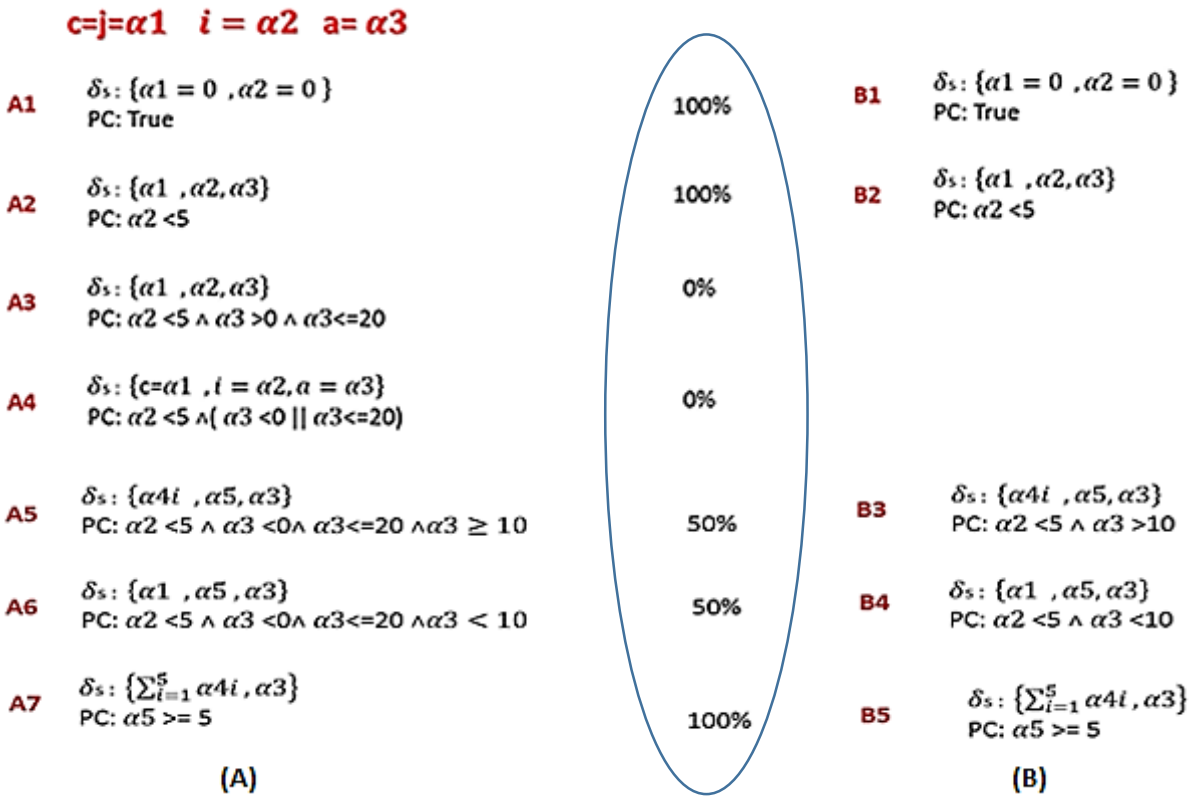


Figure 24 : Taux de ressemblance entre les deux solutions (prof et étudiant) de l'exemple 2 selon la variante A

La figure 24 (A) représente l'exécution symbolique du programme du professeur et la figure 24(B) représente l'exécution symbolique du programme de l'étudiant.

Rappelons que la méthode A, consiste à comparer les PC si et seulement si les stores relatifs aux deux blocs de base comparés sont identiques.

Par exemple entre le Bloc A5 et B3 : on relève un taux de ressemblance de 50%. En effet les stores sont les mêmes est 2 conditions / 4 conditions sont vérifiées.

La note est calculée de la manière suivante :

Note / 20 = $\sum_{i=1}^{n} \frac{pourcentage(i)}{n} * \frac{20}{100}$

= (100+100+50+50+100/7)*20/100

= 11.5/20

NB : n représente le nombre maximal des blocs de base constituant la solution modèle et le programme évalué.

✓ Calcul de similarité en utilisant la 2ème variante (variante Z) :

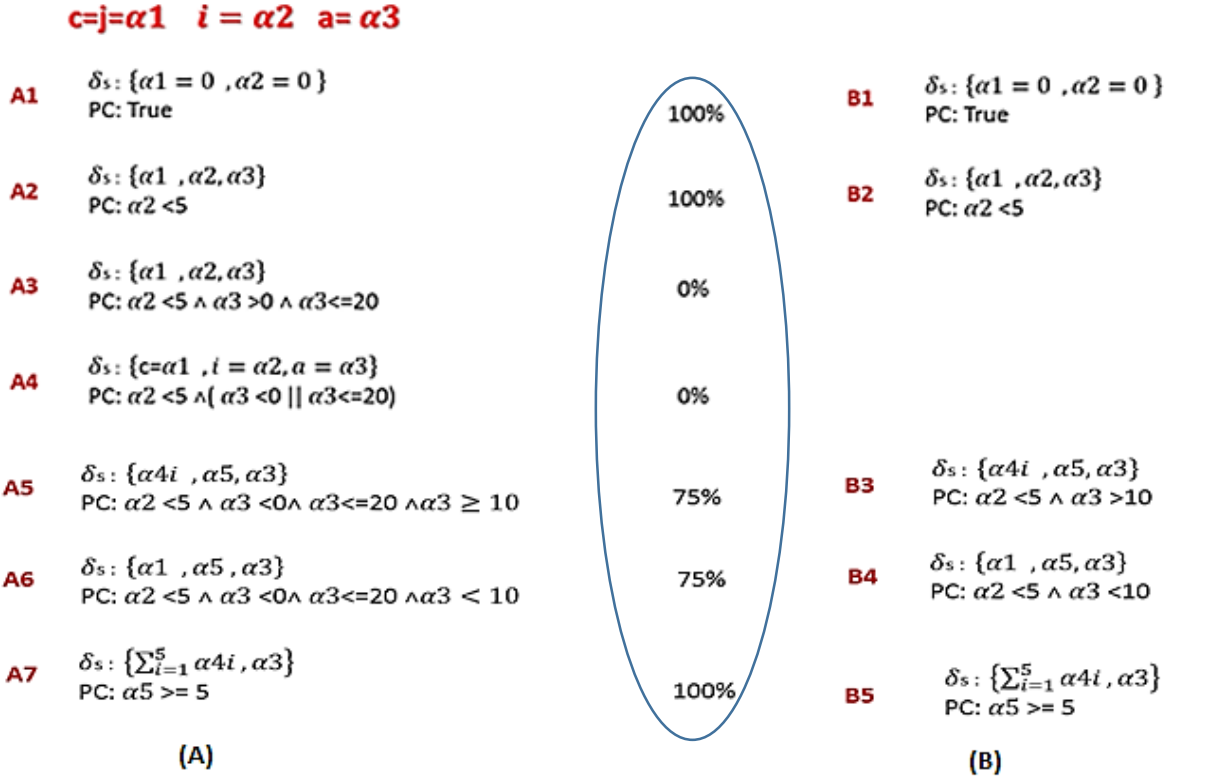


Figure 25 : Taux de ressemblance entre les deux solutions (prof et étudiant) de l'exemple 2 selon la variante Z

Rappelons que la méthode Z, consiste à attribuer 50% pour les PC et 50% pour les stores :

Note / 20 = $\sum_{i=1}^{n} \frac{pourcentage(i)}{n} * \frac{20}{100}$

= (100+100+75+75+100/7)*20/100

= 12.85/20

5.5.5 Exemple 3 : l'exécution symbolique dans le cas de (if –else):

Soit le programme qui permet de calculer les solutions d'une équation du 2ème degré.

Le programme modèle (proposé par l'évaluateur) est présenté dans la figure 25:

```
printf("Introduisez les valeurs pour a, b, et c : ");
scanf("%i %i %i", &A, &B, &C);
/* Calcul du discriminant b^2-4ac */
D = pow(B,2) - 4.0*A*C;
/* Distinction des différents cas */
if (A==0 && B==0 && C==0) /* 0x = 0 */
printf("Infinité de solution.\n");
else if (A==0 && B==0) /* Contradiction: c # 0 et c = 0 */
printf("pas de solutions.\n");
else if (A==0) /* bx + c = 0 */
{
printf("équation de 1ere degré:\n");
x = (double)C/B;
}
else if (D<0) /* b^2-4ac < 0 */
{
//printf("Les solutions complexes de cette équation sont les suivantes :\n");
x1 = (double)(-B) + i(double)(sqrt(-D)/(2*A)) ;
x2 = (double)(-B) - i(double)(sqrt(-D)/(2*A)) ;
}
else if (D==0) /* b^2-4ac = 0 */
{
//printf("Cette équation a une seule solution réelle :\n");
x1= (double)-B/(2*A);
}
else /* b^2-4ac > 0 */
{
//printf("Les solutions réelles de cette équation sont :\n");
x1=(double)(-B+sqrt(D))/(2*A));

x2=(double)(-B-sqrt(D))/(2*A));
}
```

Figure 26 : Programme associé à l'exemple 3 proposé par un professeur

On suit les mêmes étapes précédemment décrites, à savoir la visualisation des blocs de base sous forme de CFG et l'exécution symbolique des différents blocs de base.

1ère étape : Le graphe de flot de contrôle de la solution modèle est le suivant :

Chapitre 5 :l'exécution symbolique

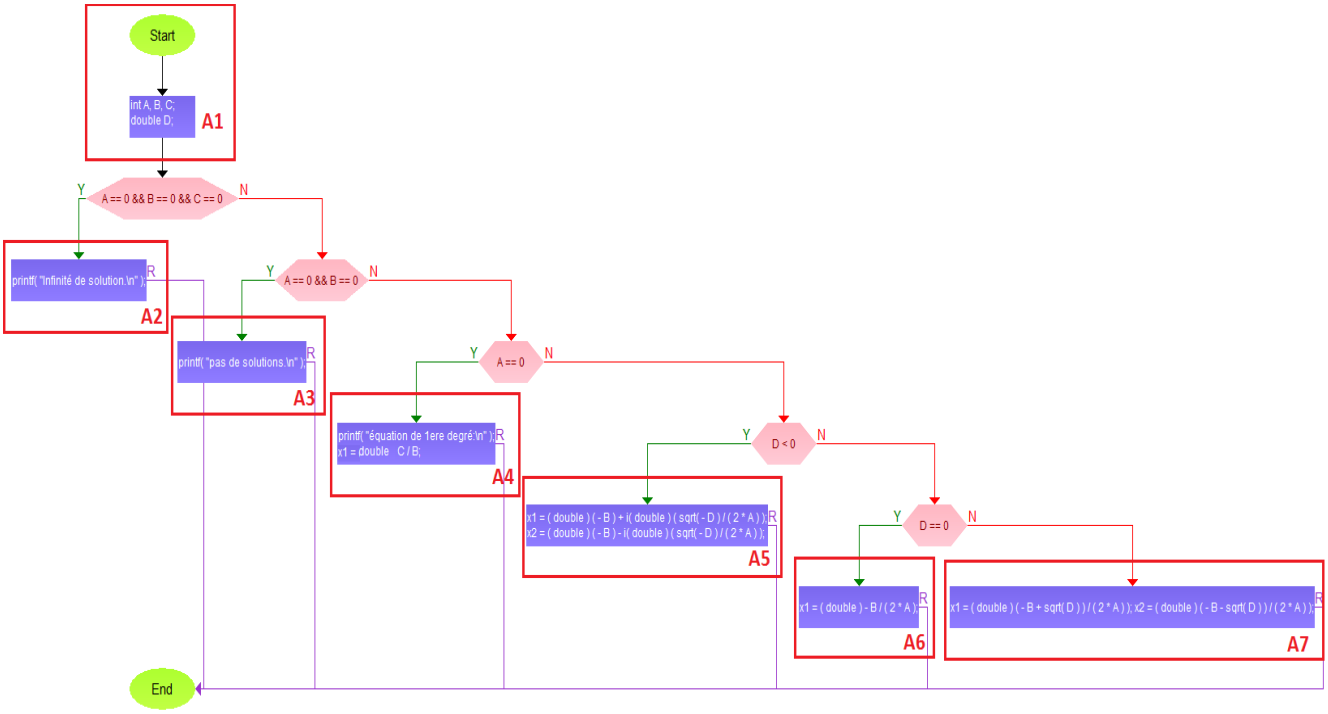


Figure 27: Graphe de flot de contrôle du programme proposé par le professeur pour l'exemple 3

2ème étape : L'exécution symbolique associée à chaque bloc de base est représentée dans la figure suivante :

A1 δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ ,\ D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3)\ , x1 = \alpha5\ , x2 = \alpha6\}$

A2 PC: $\alpha1 = 0 \wedge \alpha2 = 0 \wedge \alpha3 = 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = \alpha5, x2 = \alpha6\}$

A3 PC: $\alpha1 = 0 \wedge \alpha2 = 0 \wedge \alpha3! = 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = \alpha5, x2 = \alpha6\}$

A4 PC: $\alpha1 = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\frac{\alpha2}{\alpha3}, x2 = \alpha6\}$

A5 PC: $\alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge \alpha4 < 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, x2 = -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

A6 PC: $\alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge \alpha4 = 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2/2\alpha1, x2 = \alpha6\}$

A7 PC: $\alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge \alpha4 > 0$
δ_s: $\{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, x2 = -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

Figure 28 : L'exécution symbolique de la solution de l'exemple 2 proposée par un professeur

Chapitre 5 :l'exécution symbolique

Voici le programme proposé par un étudiant pour le calcul des solutions d'une équation du second degré :

```
main()
{
/* Calcul des solutions réelles et complexes d'une équation du second degré */
int A, B, C;
double D; /* Discriminant */
printf("Calcul des solutions réelles et complexes d'une équation du second \n");
printf("degré de la forme ax^2 + bx + c = 0 \n\n");
printf("Introduisez les valeurs pour a, b, et c : ");
scanf("%i %i %i", &A, &B, &C);
/* Calcul du discriminant b^2-4ac */
D = pow(B,2) - 4.0*A*C;
if (D>0) /* b^2-4ac > 0 */
{
//printf("Les solutions réelles de cette équation sont :\n");
x1=(double)(-B+sqrt(D))/(2*A));
x2=(double)(-B-sqrt(D))/(2*A));
}
else if (D==0) /* b^2-4ac = 0 */
{
//printf("Cette équation a une seule solution réelle :\n");
x1= (double)-B/(2*A);
}
else
{
//printf("Les solutions complexes de cette équation sont les suivantes :\n");
x1 = (double)(-B) + i(double)(sqrt(-D)/(2*A)) ;
x2 = (double)(-B) - i(double)(sqrt(-D)/(2*A)) ;

}
return 0;
}
```

Figure 29 : Programme associé à l'exemple 3 proposé par un étudiant

1ère étape : Le graphe de flot de control de la solution proposée par l'étudiant est le suivant :

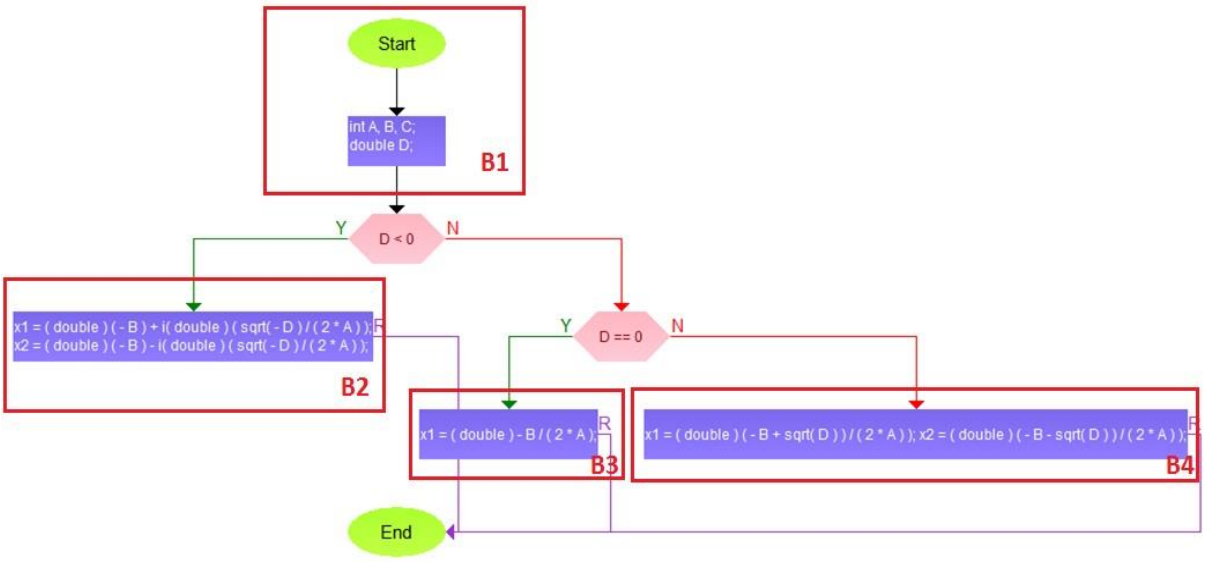


Figure 30: Graphe de flot de contrôle du programme d'étudiant de l'exemple 3

2ème étape : L'exécution symbolique associée à chaque bloc de base est représentée dans la figure 30 :

B1 $\delta_s : \{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ ,\ D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3)\ , x1 = \alpha5\ , x2 = \alpha6\}$

B2 $PC:\ \alpha4 > 0$
$\delta_s : \{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, x2 = -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

B3 $PC:\ \alpha4 = 0$
$\delta_s : \{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2/2\alpha1, x2 = \alpha6\}$

B4 $PC:\ \alpha4 < 0$
$\delta_s : \{A = \alpha1\ ,\ B = \alpha2, C{=}\alpha3\ , D{=}\alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), x1 = -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, x2 = -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

Figure 31 : l'exécution symbolique de la solution de l'exemple 3 proposée par un étudiant

✓ **Calcul de similarité en utilisant La variante A :**

A1 $\delta_s : \{\alpha1, \alpha2, \alpha3\ ,\ \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3)\ , \alpha5\ , \alpha6\}$

100%

B1 $\delta_s : \{\alpha1, \alpha2, \alpha3\ ,\ \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3)\ , \alpha5\ , \alpha6\}$

A7 $PC:\ \alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge\ \alpha4 > 0$
$\delta_s : \{\alpha1\ ,\ \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

25%

B2 $PC:\ \alpha4 > 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

A6 $PC:\ \alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge\ \alpha4 = 0$
$\delta_s : \{\alpha1\ ,\ \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2/2\alpha1, \alpha6\}$

25%

B3 $PC:\ \alpha4 = 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2/2\alpha1, \alpha6\}$

A5 $PC:\ \alpha1! = 0 \wedge \alpha2! = 0 \wedge \alpha3! = 0 \wedge\ \alpha4 < 0$
$\delta_s : \{\alpha1\ ,\ \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

25%

B4 $PC:\ \alpha4 < 0$
$\delta_s : \{\alpha1\ ,\ \alpha2, \alpha3\ , \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

Figure 32: Taux de ressemblance entre les deux solutions (prof et étudiant) de l'exemple 2 selon la variante A

La note de l'étudiant est calculée en se basant sur la formule suivante :

Note / 20 = $\sum_{i=1}^{n} \frac{pourcentage(i)}{n} * \frac{20}{100}$

Note / 20 = 5 /20

✓ **Calcul de similarité en utilisant La variante Z :**

A1 $\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), \alpha5, \alpha6\}$

100%

B1 $\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), \alpha5, \alpha6\}$

A7 $PC: \alpha1 != 0 \wedge \alpha2 != 0 \wedge \alpha3 != 0 \wedge \alpha4 > 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

62,5%

B2 $PC: \alpha4 > 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \frac{\sqrt{\alpha4}}{2\alpha1}, -\alpha2 - \frac{\sqrt{\alpha4}}{2\alpha1}\}$

A6 $PC: \alpha1 != 0 \wedge \alpha2 != 0 \wedge \alpha3 != 0 \wedge \alpha4 = 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2/2\alpha1, \alpha6\}$

62,5%

B3 $PC: \alpha4 = 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2/2\alpha1, \alpha6\}$

A5 $PC: \alpha1 != 0 \wedge \alpha2 != 0 \wedge \alpha3 != 0 \wedge \alpha4 < 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

62,5%

B4 $PC: \alpha4 < 0$
$\delta_s : \{\alpha1, \alpha2, \alpha3, \alpha4 = (\alpha2^2 - 4 * \alpha1 * \alpha3), -\alpha2 + \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i, -\alpha2 - \left(\frac{\sqrt{-\alpha4}}{2\alpha1}\right) i\}$

Figure 33 : Taux de ressemblance entre les deux solutions (prof et étudiant) de l'exemple 2 selon la variante Z

La note de l'étudiant en utilisant la variante Z :

Note / 20 = $\sum_{i=1}^{n} \frac{pourcentage(i)}{n} * \frac{20}{100}$

Note / 20 = 8.21 /20

5.5.6 Comparaison de Méthode A, la méthode Z et la notation manuelle

Afin d'évaluer l'efficacité des variantes proposées, une comparaison avec la notation manuelle s'est avérée intéressante.

Chapitre 5 :l'exécution symbolique

Tableau 1 : Tableau comparatif entre les notes de la variante A, La variante Z, et la notation manuelle.

Solution	variante A	variante Z	Notation Manuelle
Exemple 1 Solution 2	0	4,5	7
Exemple 2 Solution 1	11,5	12,85	10
Exemple 2 Solution 2	12,5	13,21	15
Exemple 3 Solution 1	5	8,21	11.5
Exemple 4 Solution 1	0	3,5	5
Exemple 4 Solution 2	9	9	12,5

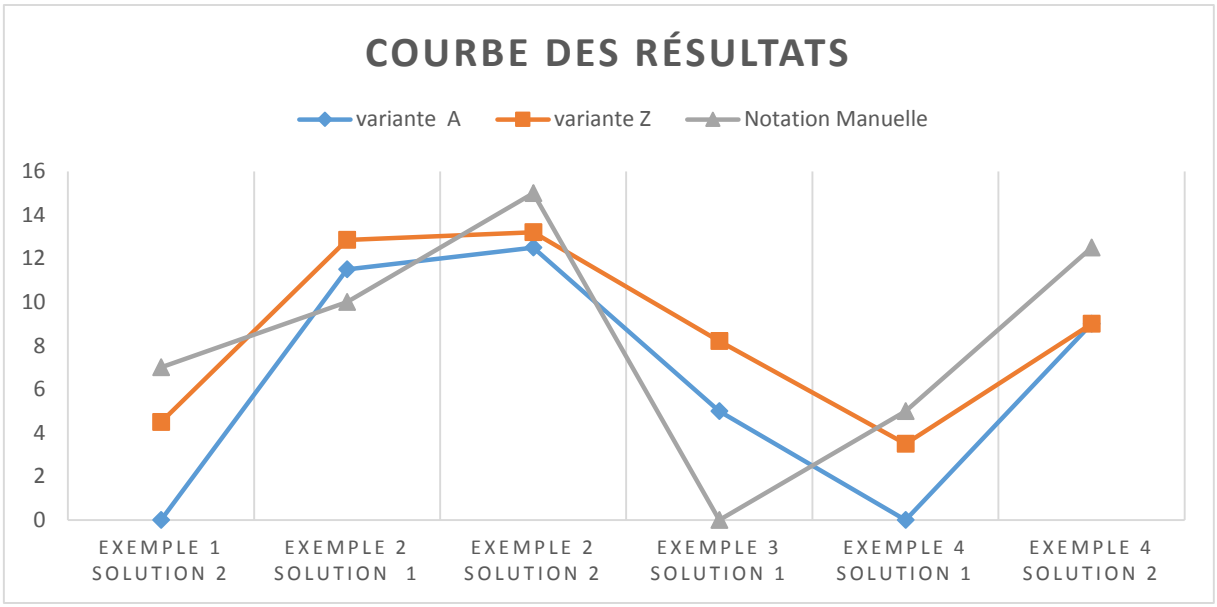


Figure 34 : Courbe des notes obtenue en appliquant les variantes (Z et A) et notation manuelles

En général les trois courbes représentant la variante A, la variante Z et l'évaluation manuelle ont la même tendance. Cependant, on remarque que la variante Z est plus proche de la notation manuelle.

5.5.7 Calcul de similarité en faisant appel à la pondération au niveau des blocs de base

Afin de garantir une approximation maximale aux résultats de la notation manuelle, une nouvelle variante a été proposée à travers la pondération des blocs de base. Cette variante consiste à associer un poids à chaque bloc de base. Cela permet de valoriser certains blocs de base par rapport aux autres. Dans ce cas, la formule de calcul de similarité serait comme suit :

$$\text{Note} = \left(\sum_{i=1}^{n}(\boldsymbol{poids(i)} * \boldsymbol{pourcentage(i)})\ /\mathbf{100})\right) * \frac{\mathbf{20}}{\mathbf{100}}$$

Nb : $\sum_{i=1}^{n} \boldsymbol{poids(i)} = \mathbf{100}\%$

Le taux de ressemblance entre les blocs de base de l'exemple 3 en utilisant la variante de pondération :

Les poids ont été attribués aux différents blocs de base de la manière suivante :

Poids(A5) = Poids(A6)= Poids(A7)= Poids(A1)=20%

Poids(A2)+Poids(A3)+Poids(A4)=20%

Les blocs de base A5, A6, A7 et A1 ont le même poids égal à 20%, soit le total des poids des 4 blocs de base égal à 80%.

La somme des poids des blocs de base A2, A3 et A4 est égale à 20%.

✓ Exemple 3 : La variante A + pondération :

En appliquant le principe de la pondération à la variante A, la note sera calculée comme suit :

Note = (25*20 +25*20+25*20+100*20)/100 *20/100

Note = 7/20

✓ Exemple 3 : La variante Z + pondération :

En appliquant le principe de la pondération à la variante Z, la note sera calculée comme suit :

Note = (65,5*20+65,5*20+65*20+65,5*20)/100*20/100)

Note =11,5/20

En se référant aux résultats obtenus à travers l'application de la variante de pondération aux deux variantes A et Z, on constate que la note obtenue par la variante Z avec pondération des blocs de base est similaire à la note obtenue par la notation manuelle. Cette constatation a été confirmée à travers d'autres exemples de programmes. Cela implique que la variante la plus proche de l'évaluation humaine est la variante intitulée Z+pondération. Nous rappelons que cette variante considère à parts égales le store des données et le Path Condition dans la comparaison des blocs de base des programmes en question. S'ajoute à cela l'affectation de poids différents aux blocs de base dans le calcul de similarité.

Chapitre 5 :l'exécution symbolique

Vu les résultats considérables obtenus par l'application de la variante Z+ pondération, cette dernière sera adoptée par notre outil d'évaluation automatique qu'on détaillera dans le chapitre suivant

Chapitre 6 : Application

6. Application

6.1 Outils et méthodologie

Actuellement, plusieurs grandes plates-formes existent sur le marché. Elles sont globalement constituées de deux composantes : le langage de programmation et la base de données. Nous citons ici une liste non exhaustive des différents outils utilisés :

6.1.1 langage de programmation

Pour l'implémentation des différents modules de l'application, nous avons utilisé le langage de **programmation PHP**.

PHP : c'est un langage incrusté au HTML et interprété (PHP3) ou compilé (PHP4) côté serveur. Il dérive du langage C et du Perl dont il reprend la syntaxe. Il est extensible grâce à de nombreux modules et son code source est ouvert. Comme il supporte tous les standards du web et qu'il est gratuit, il s'est rapidement répandu sur la toile.

Il est caractérisé par sa portabilité et utilisé pour la programmation orientée objet.

Le langage **JAVASCRIPT** qui est interprété par le navigateur au moment du chargement du document est utilisé ici pour le contrôle des valeurs de champs d'un formulaire et également pour la gestion des évènements.

C'est un langage orienté objet permettant d'exploiter la structure des données mise en place par le navigateur lorsque celui-ci charge un document HTML.

Le langage **HTML et le CSS (feuilles de style)** ont été utilisés pour la description et le design de nos différentes interfaces.

6.1.2 Bases de données

Comme SGBD, nous avons utilisé MySQL qui est un serveur de bases de données relationnelles SQL développé dans un souci de performances élevées en lecture, ce qui signifie qu'il est davantage orienté vers le service de données déjà en place que vers celui de mises à jour fréquentes et fortement sécurisées. Il est multithread et multiutilisateurs.

6.1.3 Méthodes et logiciels employés

Pour modéliser notre application, nous avons utilisé le langage **UML** (en anglais ***Unified Modeling Language***, « langage de modélisation unifié »). C'est un langage graphique de modélisation des données et des traitements. UML n'impose pas une méthode de travail particulière, il peut donc être intégré à n'importe quel processus de développement logiciel de manière transparente. C'est une sorte de boîte à outils, qui permet d'améliorer progressivement les méthodes de travail, tout en préservant les modes de fonctionnement. Intégrer UML par étapes dans un processus, de manière pragmatique, est tout à fait possible.

Pour la modélisation des diagrammes UML, nous nous sommes servis d'**Enterprise Architect** qui joue le rôle d'un AGL (Atelier de génie logiciel) de conception. Les AGL de Conception aident à la réalisation de la phase de conception et d'analyse.

Chapitre 6 : Application

Pour la programmation des différents modules de l'application, nous avons utilisé **Notepad++**qui est un environnement de développement intégré (IDE) pour **PHP, HTML, JAVASCRIPT et AJAX.**

WampServer est une plate-forme de développement Web sous Windows pour des applications Web dynamiques à l'aide du serveur Apache2, du langage de scripts PHP et d'une base de données MySQL. Il possède également PHPMyAdmin pour gérer plus facilement les bases de données.

Comme navigateurs, nous avons utilisé **MOZILLA FIREFOX** et **GOOGLE CHROME.**

6.2 Analyse et conception :

Pour mener à bien le projet, nous devons tout naturellement avoir recours à un formalisme de conception à savoir **UML « Unified Modeling Language »** qui est le langage de modélisation graphique permettant de comprendre et de décrire les besoins, de spécifier et documenter le système ainsi que d'esquisser les architectures logicielles.

Dans cette section, nous présentons quelques diagrammes de modélisation en explicitant leurs rôles. Nous allons principalement présenter le diagramme des cas d'utilisation et le diagramme de classes.

6.2.1 Diagramme de cas d'utilisation:

Définition :

Les cas d'utilisations permettent de structurer les besoins des utilisateurs et les objectifs correspondants d'un système. Ils centrent l'expression des exigences du système sur ses utilisateurs en clarifiant et en organisant leurs besoins (les modéliser).

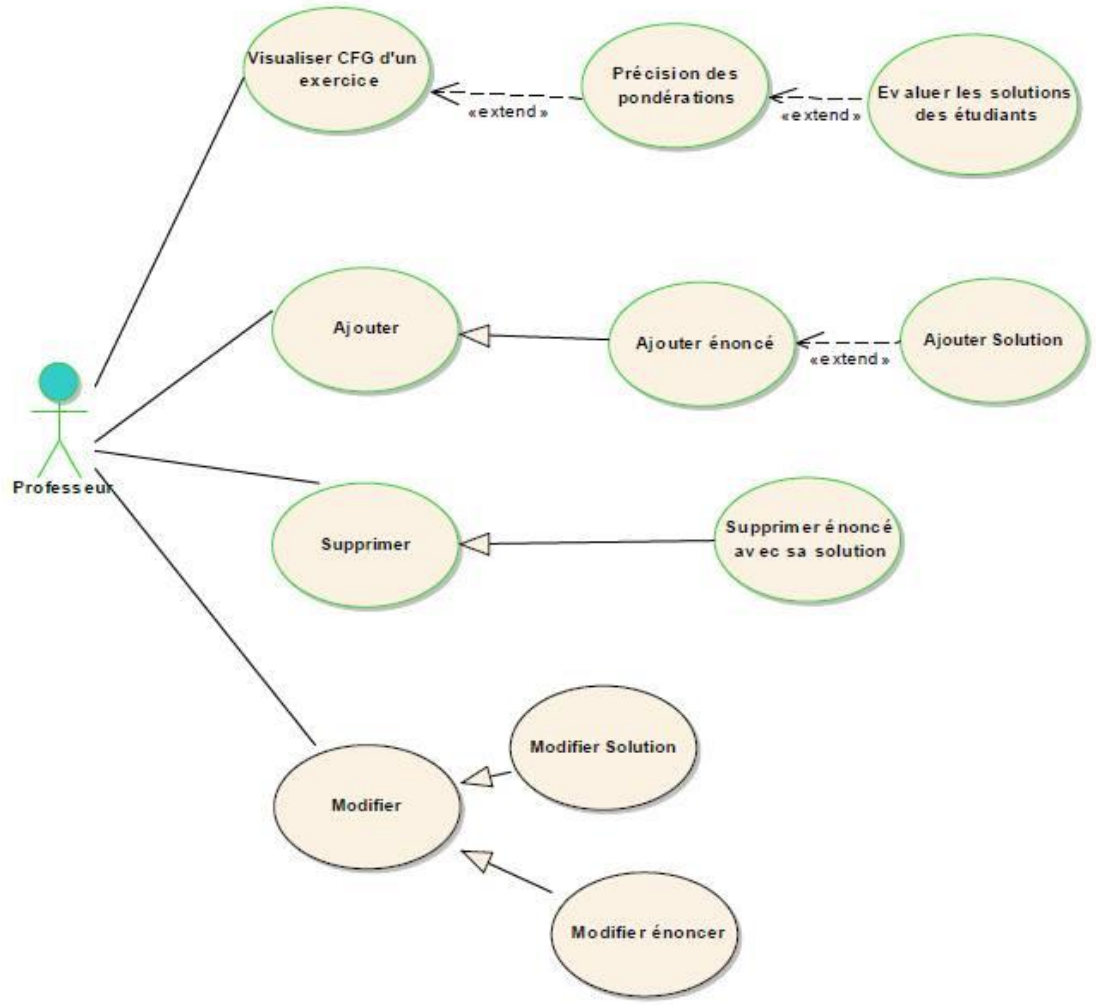


Figure 35 : Diagramme de cas d'utilisation Professeur :

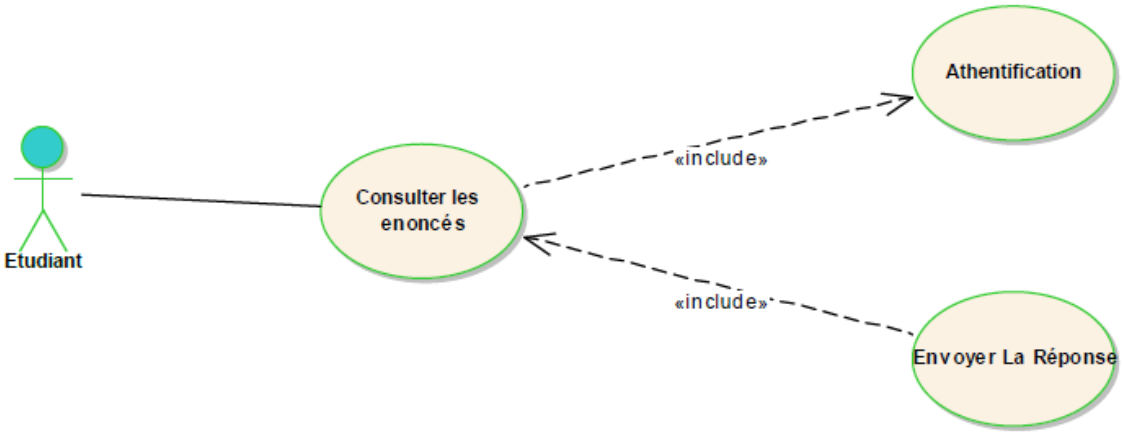


Figure 36: Diagramme de cas d'utilisation Etudiant

6.2.2 Diagramme de classes:

Définition :

Il représente les classes intervenant dans le système. Le diagramme de classe est une représentation statique des éléments qui composent un système et de leurs relations.

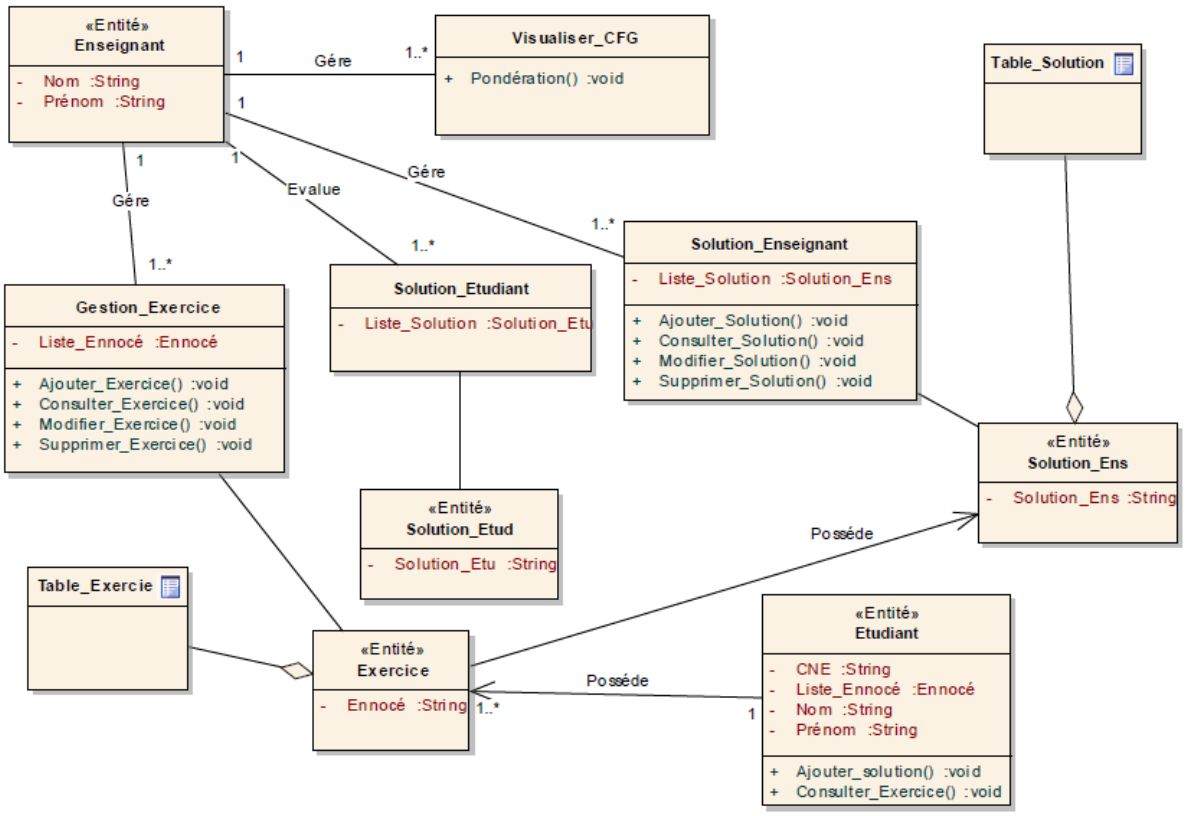


Figure 37: Diagramme de classe

Chapitre 6 : Application

6.3 L'application développée

6.3.1 Interface étudiant

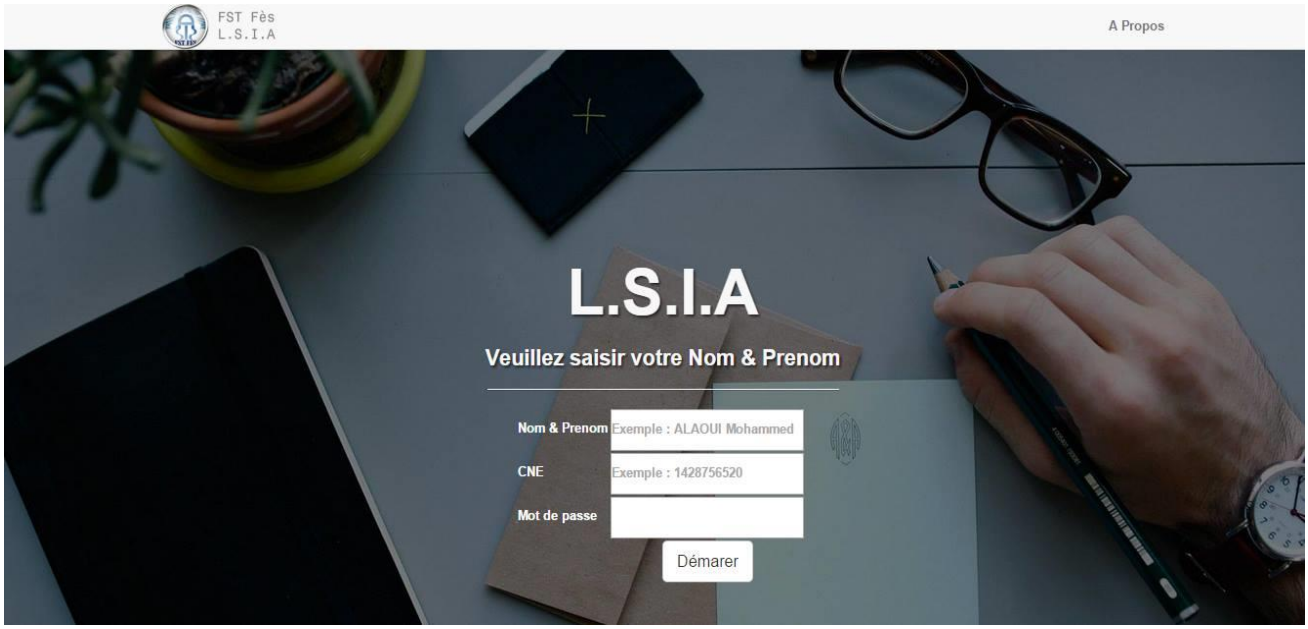


Figure 38 : page d'accueil de l'interface étudiant

Avant d'accéder à l'application pour soumettre les solutions des exercices l'étudiant doit entrer son nom et son prénom son CNE et son mot de passe pour l'authentification.

Figure 39: La page des exercices à résoudre

Cette page contient les exercices à résoudre proposé par le professeur il peut afficher les exercices afin de soumettre la solution associé à chaque exercice.

Chapitre 6 : Application

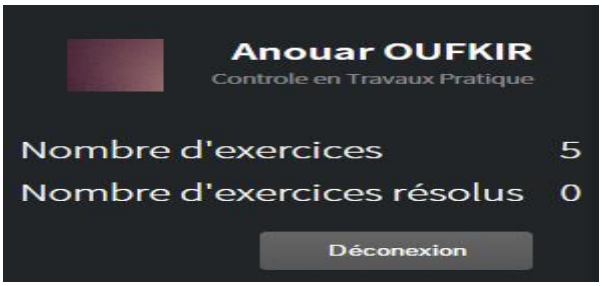


Figure 40 : menu nombre des exercices résolus et non résolus

Dans le menu à gauche de la page exercices à résoudre il y a le nombre des exercices à résoudre ainsi que le nombre d'exercice résolu.

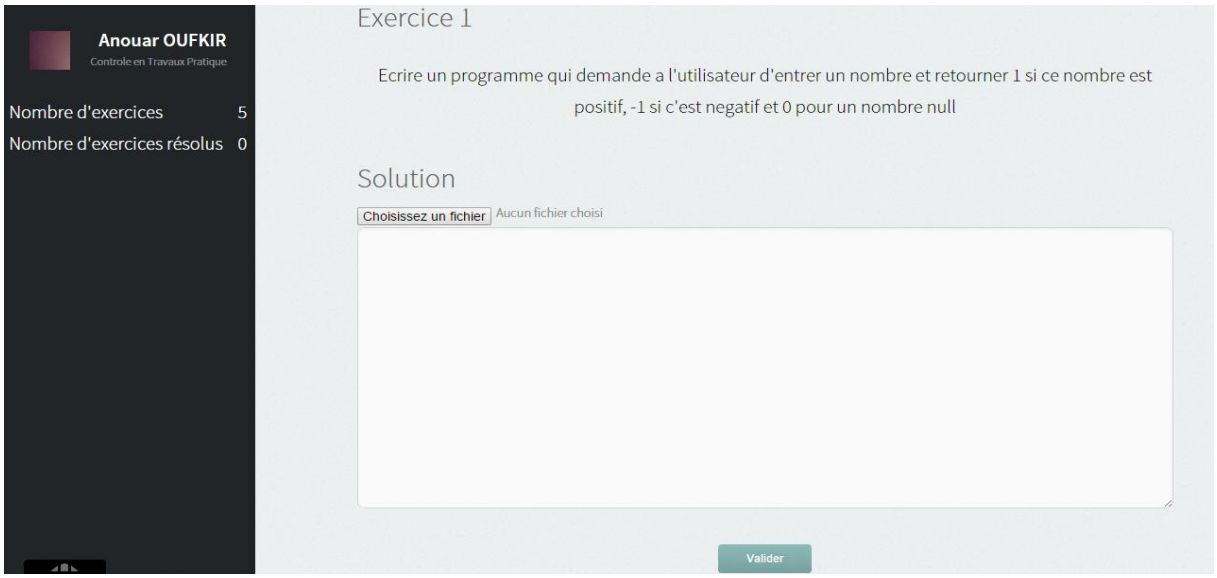


Figure 41 : page de la résolution d'un exercice par un étudiant

Dans la page exercices à résoudre figure 39, lorsqu'on clique sur afficher l'énoncé de l'exercice, ce dernier s'affiche et on peut ajouter une solution en insérant la solution ou en exportant la solution.

Après la validation on obtient :

Figure 42: page de validation d'une solution d'un exercice

Chapitre 6 : Application

Le nombre des exercices résolu s'incrémente lors de l'ajout d'une nouvelle solution et chaque exercice résolu ne peut pas être résolu encore une fois et s'enregistre et s'affiche de la manière suivante :

Figure 43: solution enregistrée

6.3.2 Interface professeur

Figure 44: page d'accueil professeur

Dans la page d'accueil on trouve le menu en haut permettant de gérer les exercices ou d'évaluer les exercices envoyés par les étudiants.

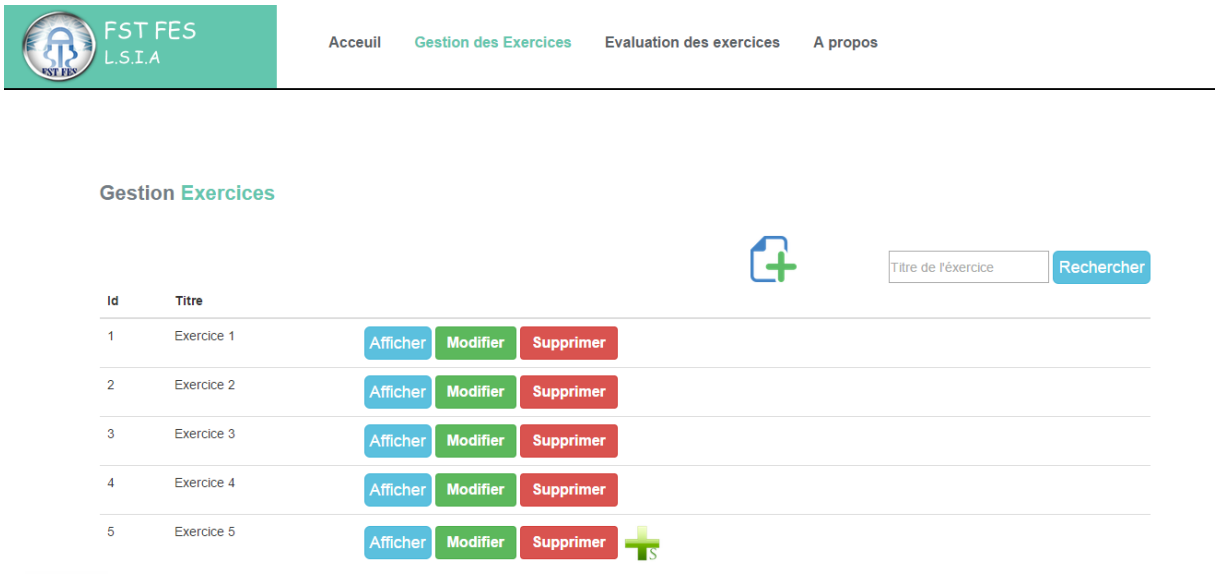


Figure 45:page gestion des exercices

Chapitre 6 : Application

La page de gestion des exercices consiste à ajouter un exercice et la solution proposée par le professeur, de supprimer des exercices et leurs solutions, de modifier des exercices et leurs solutions et rechercher un exercice à travers le nom de l'exercice et aussi afficher l'énoncé et la solution d'un exercice.

Pour ajouter un exercice on clique sur l'icône suivante :

Figure 46: icône d'ajout d'un exercice

Lorsqu'on appuie sur l'icône le formulaire d'ajout s'affiche :

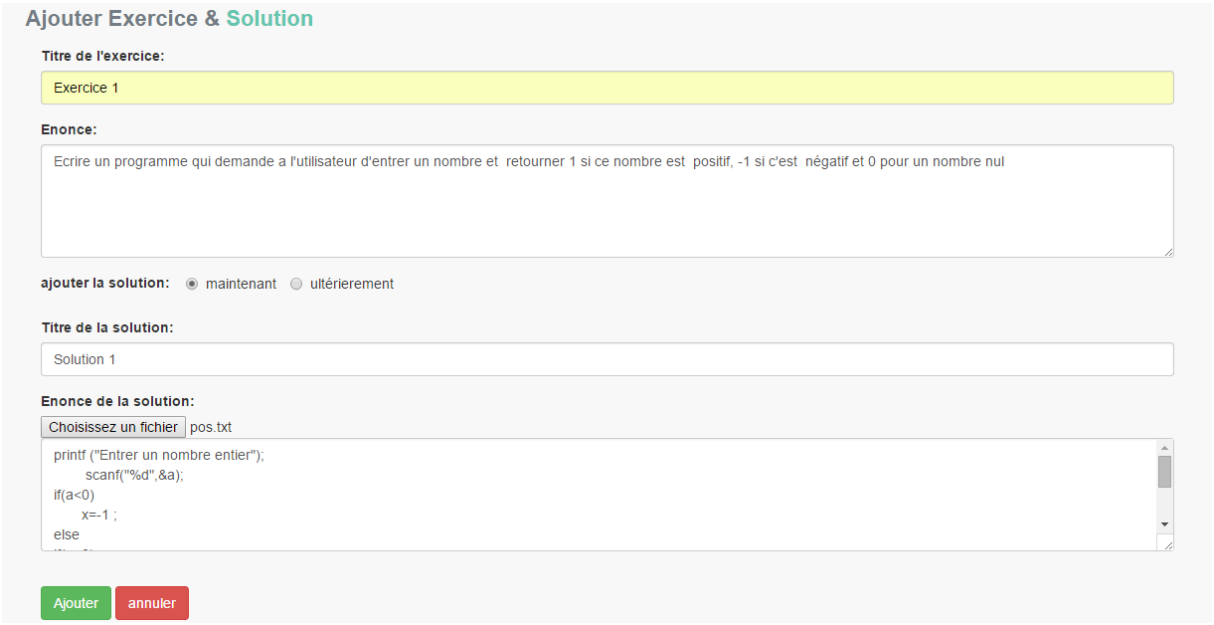


Figure 47: formulaire d'ajout d'un exercice

Le professeur peut ajouter uniquement l'énoncé d'un exercice ou l'énoncé et la solution associé à cet exercice proposé par le professeur.

Pour les exercices existant sans solution on peut ajouter leurs solutions à travers le bouton suivant qui se trouve dans la page gestion des exercices figure 45

Figure 48:bouton d'ajout d'une solution

Chapitre 6 : Application

On peut afficher l'énoncé d'un exercice ainsi que sa solution en appuyant sur le bouton afficher de la page gestion des exercices figure 45.

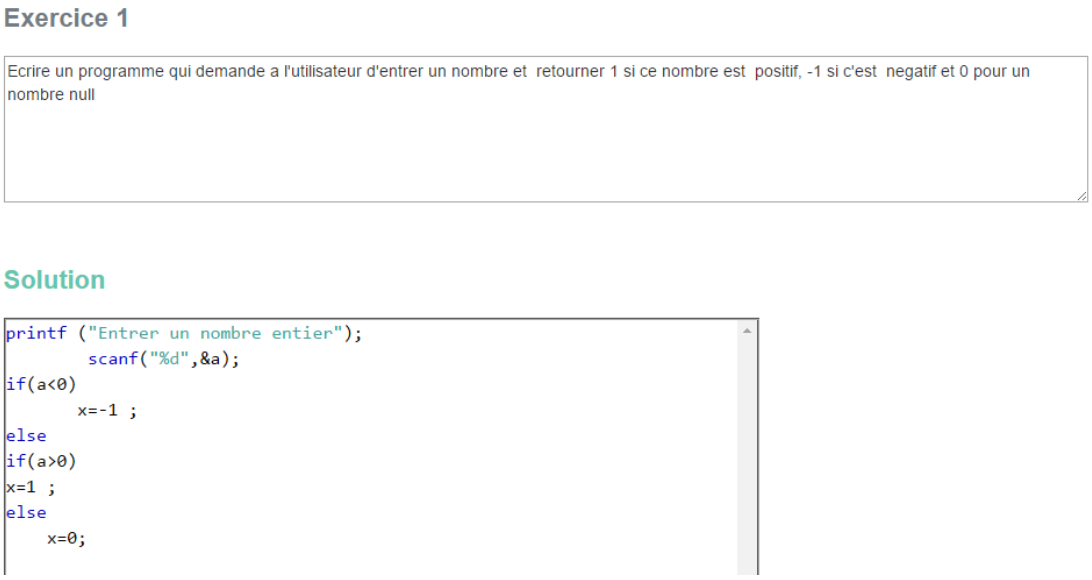


Figure 49 : page de l'affichage d'un exercice et sa solution

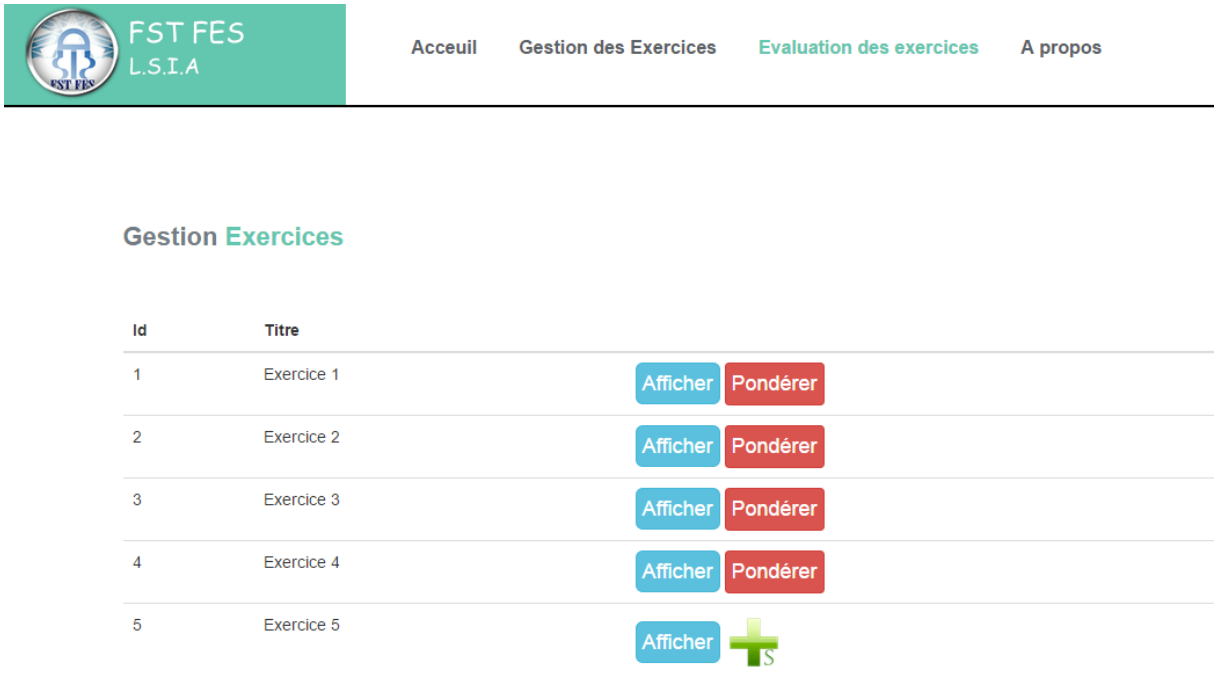


Figure 50: page d'évaluation d'un exercice

Dans la page d'évaluation d'un exercice on peut afficher un exercice et sa solution le bouton pondérer permet d'associé un pourcentage à chaque bloc de base.

Dans cette page on peut aussi ajouter une solution pour des exercices sans solutions.

Chapitre 6 : Application

Lorsqu'on appuie sur le bouton pondérer le graphe de flot de contrôle s'affiche découpé sous forme de bloc de base contenant des champs de saisi permettant d'attribuer un pourcentage de pondération manuelle en entrant des pourcentages ou de manière automatique dans le cas des pourcentages égales en appuyant sur le bouton générer le pourcentage

Le bouton valider permet d'enregistrer les pondérations dans la base de données.

Graphe de flot de contrôle

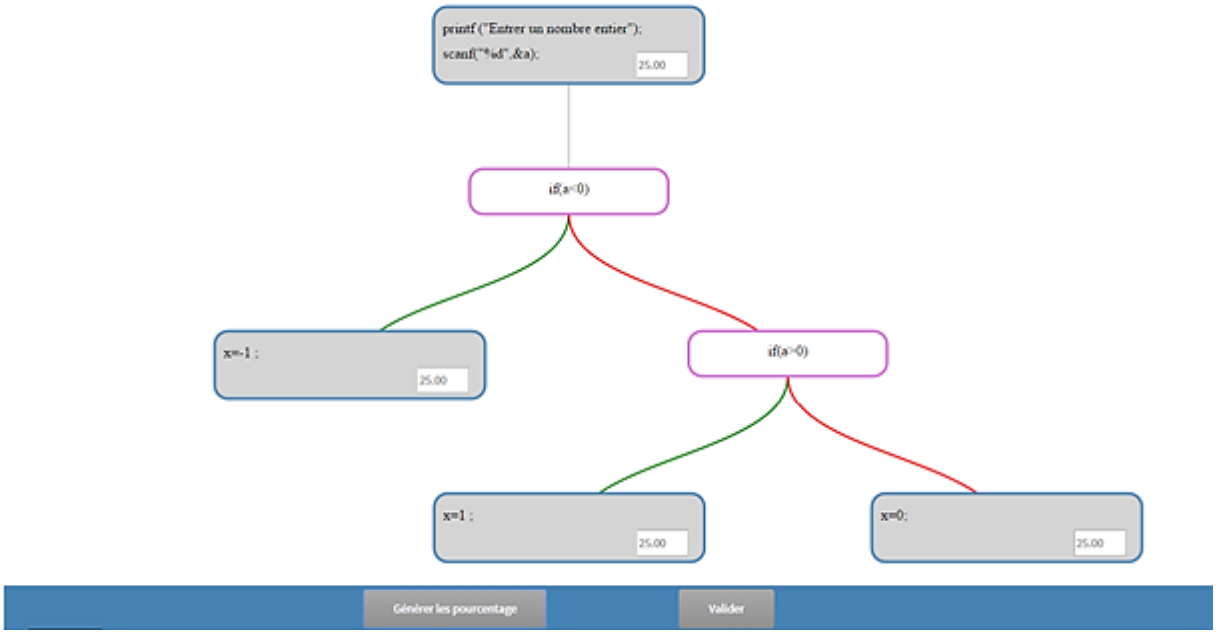


Figure 51: page de pondération de blocs de base

Une fois la pondération est enregistré le bouton évalué s'affiche qui permet d'évaluer les solutions de l'exercice choisi.

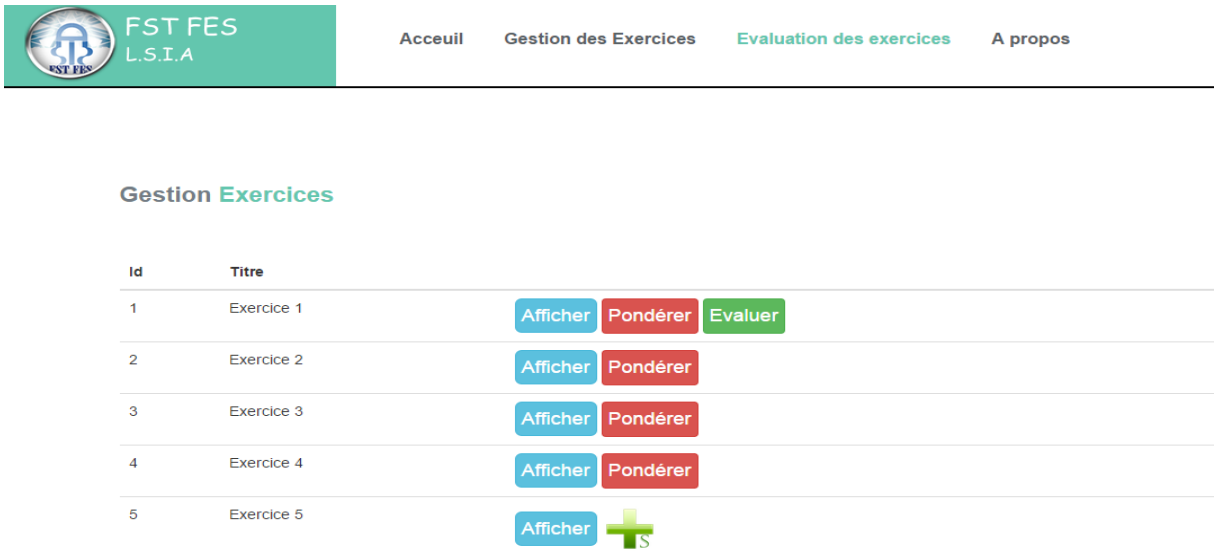


Figure 52: affichage du bouton évaluer en cas d'existence de la pondération d'un exercice

Chapitre 6 : Application

Figure 53: page évaluer un exercice

Après avoir appuyé sur le bouton évaluer les notes des étudiants associés à l'exercice choisi s'affichent.

Dans la page évaluer on peut afficher la solution d'étudiant et de professeur ou de visualiser la solution de l'étudiant sous forme de graphe de flot de contrôle.

Figure 54: affichage de la solution étudiant et professeur

Chapitre 6 : Application

Graphe de flot de contrôle

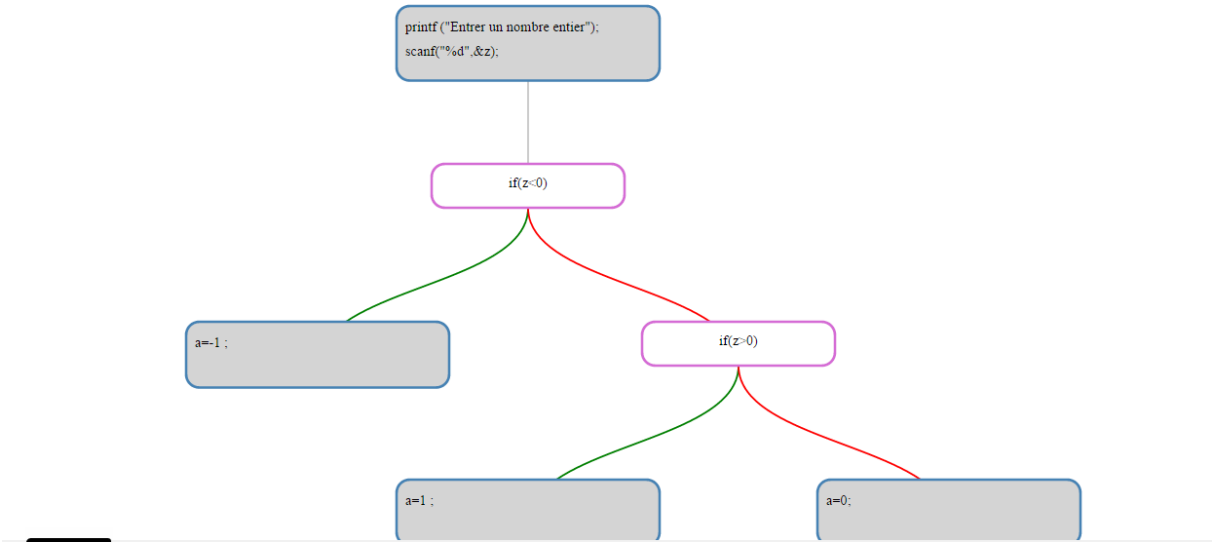


Figure 55: visualiser la solution de l'étudiant sous forme de CFG

Conclusion

L'intégration des systèmes d'évaluation automatiques des programmes dans les cours d'informatique contribue à la réduction de la charge de correction et d'évaluation manuelle des programmes, en effet, l'évaluation manuelle est toujours une tâche banale qui nécessite souvent beaucoup de temps et d'efforts.

L'avantage majeur de l'analyse statique réside dans la possibilité d'analyser et d'examiner un programme sans l'exécuter, cela justifie notre choix pour l'utilisation de l'une de ses méthodes qui considère la sémantique des programmes.

Durant nos recherches à propos de la similarité sémantique une nouvelle approche qui s'appuie sur l'exécution symbolique a été proposée.

L'exécution symbolique est une méthode d'analyse statique permettant de prouver certaines propriétés du programme à savoir les données et les conditions.

La similarité sémantique à travers l'exécution symbolique nous a permis de noter les exercices des étudiants en comparant les exécutions symboliques de chaque bloc de base de la solution modèle ou solution correcte avec l'exécution symbolique des blocs de base des solutions fournie par les étudiants .

Trois variantes ont été envisagées dans le cadre de la notation des exercices à travers l'exécution symbolique, la méthode la plus adaptée et la plus pertinente se présente dans la concaténation de la variante Z et la pondération des blocs de base, cette dernière a pu fournir des résultats satisfaisants et très proches de la notation manuelle, cela a été prouvé par les résultats obtenues en utilisant cette variante pour le calcul du taux de similarité entre programmes dans l'application développée.

Au cours de l'implémentation de la variante choisie dans notre application, le recours à la normalisation des conditions pour la comparaison des Paths Condition s'est avéré indispensable. Cela constitue une tache laborieuse et qui implique l'invocation d'un grand nombre de règles à appliquer.

Cette application nous a permis d'évaluer des programmes de petites tailles de complexités moyennes écrits par des étudiants novices en programmation.

Finalement, en guise que perceptives, nous pensons à améliorer notre application par l'implémentation du maximum de règles de normalisation afin de prévoir un système fonctionnel pour tout programme.

Référence:

[1] Rahman, K. A., & Nordin, M. J. (2007, December). A review on the static analysis approach in the automated programming assessment systems. InProceedings of the national conference on programming (Vol. 7).

[2] Caiza, J. C., & Ramiro, Á. (2013). Programming assignments automatic grading: review of tools and implementations.

[3] Jean-Yves Bouttery (2007) .Test et Validation du logiciel

[4] Vujošević-Janičić, M., Nikolić, M., Tošić, D., & Kuncak, V. (2013). Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology, 55(6), 1004-1016.

[5] Naudé, K. A., Greyling, J. H., & Vogts, D. (2010). Marking student programs using graph similarity. Computers & Education, 54(2), 545-561.

[6] Naudé, K. A. (2007). Assessing program code through static structural similarity (Doctoral dissertation, Nelson Mandela Metropolitan University).

[7] Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. Information and Software Technology, 49(2), 99-107.

[8] Vidal, G. (2015). Symbolic execution as a basis for termination analysis. Science of Computer Programming, 102, 142-157.

[9] Darringer, J. A., & King, J. C. (1978). Applications of symbolic execution to program testing. Computer, (4), 51-60..

[10] CADAR, Cristian et SEN, Koushik. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013, vol. 56, no 2, p. 82-90.

[11] KRINKE, Jens. Identifying similar code with program dependence graphs. In :Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. IEEE, 2001. p. 301-309.

[12] Horwitz, S., Prins, J., & Reps, T. (1988, January). On the adequacy of program dependence graphs for representing programs. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 146-157). ACM.

[13] BAXTER, Ira D., YAHIN, Andrew, MOURA, Leonardo, et al. Clone detection using abstract syntax trees. In : Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998. p. 368-377.

[14] Li, J., Pan, W., Zhang, R., Chen, F., Nie, S., & He, X. (2010). Design and implementation of semantic matching based automatic scoring system for C programming language. In Entertainment for Education. Digital Techniques and Systems (pp. 247-257). Springer Berlin Heidelberg.

[15]Bruno Dufour. Analyse et compréhension de programme.