الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
جامعة عين تموشنت بلحاج بوشعيب
Université Belahdj Bouchaib Ain Temouchent
Faculté des Sciences et Technologies
Département Mathématiques et Informatique

Projet de Fin d'Etudes

Pour l'obtention du diplôme de Master en : Informatique
Domaine: Mathématiques et Informatique
Filière : Informatique
Spécialité: Réseau et Ingénierie des Données (RID)

Thème

Analyse Logique des Données pour détecter des fautes dans les programmes

Présenté par :

- Mr Sidnas Abderrahmane
- Mr Salhi Soufiane

Devant le jury composé de :

Dr Benomar Mohammed Lamine	MCA	UAT.B.B (Ain Temouchent)	Président
Dr Mohamed Amine Messaoudi	MCB	UAT.B.B (Ain Temouchent)	Examinateur
Dr Belgrana fatima zohra	MCA	UAT.B.B (Ain Temouchent)	Encadrante
Pr Lakhdar SAIS	Professeur	Université d'Artois (Calais)	Co-Encadrant

Année Universitaire 2023/2024

Remerciement

Avant tout, nous tenons à exprimer notre profonde gratitude au Tout-Puissant, Allah, qui nous a accordé la force, la sagesse et la persévérance nécessaires pour mener à bien ce projet. Nous implorons sa bénédiction et sa guidance pour la suite de notre parcours.

Nous adressons nos plus sincères remerciements à nos chers parents, qui ont toujours été présents pour nous soutenir et nous encourager, tout au long de notre vie et de nos études. Leur amour inconditionnel et leur foi en nous ont été une source d'inspiration inestimable.

Nous tenons à remercier chaleureusement Docteur F. Z Belgrana et le professeur L.Saïs, pour leur disponibilité, leurs précieux conseils et leurs remarques constructives qui ont contribué à l'amélioration de la qualité de ce travail. Leur expertise et leur bienveillance ont été déterminantes pour notre progression.

Nous exprimons notre profonde reconnaissance aux membres du jury Docteur Benomar Mohammed Lamine et le Docteur Mohamed Amine Messaoudi d'avoir accepté de juger ce travail. Leur présence et leurs commentaires éclairés nous honorent et nous motivent à poursuivre nos efforts.

Nous remercions chaleureusement le professeur Nadjib Lazaar, ainsi que toutes les personnes qui ont contribué de près ou de loin à l'élaboration de ce modeste travail. Votre soutien et vos encouragements nous ont été précieux.

Nous adressons nos vifs remerciements à tous nos enseignants de la Faculté des Sciences et Technologies de Université Belhadj Bouchaib Ain Temouchent pour la formation de qualité qu'ils nous ont dispensée tout au long de notre cursus universitaire. Leur savoir-faire et leur passion pour leur discipline nous ont inspirés et enrichis.

Sommaire

Liste des Tableaux

Liste des Figures

Liste des Algorithmes

Liste des Acronymes

Table 1: Acronymes et Descriptions

Acronyme	Description
LF	Localisation de Faute
ALD	Analyse Logique de donnée
AA	Apprentissage Automatique
RNA	Réseaux de Neurones Artificiel
MVS	Machines à Vecteurs de Support
ACF	Analyse de Concepts Formel
AD	Arbres de Décision
MS	Mesur de Suspicion

List of Acronyms

Table 2: Acronyms and Description

Acronym	Description
SBFL	Spectrum-Based Fault localization
ML	Machine Learning
DM	Data Mining
TC	Test Case
MHS	Minimal Hitting Set
SPIC	Spanned Patterns via Input Consensus
GP13	Genetic Programming 13
Crosstab	Cross-Table
WPF	Windows Presentation Foundation
API	Application Programming Interface
ISSTA13	International Symposium on Software Testing and Analysis,2013

Introduction Générale

Malgré des efforts de test considérables d'un programme, l'apparition et la correction continue de défauts logiciels tout au long de son cycle de développement semblent être une réalité inévitable. Pour améliorer la qualité d'un programme, nous devons nous efforcer d'éliminer autant de défauts que possible sans introduire simultanément de nouveaux bugs.

Lors du débogage d'un programme, la localisation des fautes consiste à identifier les emplacements précis des erreurs dans le code. Ce processus, connu sous le nom de "problème de localisation des fautes", s'avère être une tâche extrêmement chronophage et fastidieuse. L'efficacité de la résolution de ce problème dépend de plusieurs facteurs, tels que la compréhension du programme à déboguer par le développeur, son expérience passée en débogage et la manière dont il identifie et priorise le code suspect pour l'examen des emplacements de fautes potentiels.

Notre objectif est de réaliser la localisation des fautes dans les programmes, tout en améliorant les performances. Ceci est extrêmement important dans les situations où le logiciel est embarqué ou déployé dans une application sensible comme le domaine médical par exemple.

Il existe de nombreuses approches dans littératures dans ce contexte-là. Tout au long de ce travail, nous adoptons nos approches basées sur la fouille de données.

Nous avons opté pour l'Analyse Logique des Données (ALD , ou LAD pour Logical Data Analysis). C'est une méthode qui traite les observations « positives » et « négatives », chacune étant représentée par un vecteur de n valeurs d'attributs. Ce qui est adéquat avec le type de données de notre problème. Ou les instructions non exécutées représentent des observations négatives alors que celles exécutées représentent les observations positives.

Il existe plusieurs avantages de nos approches ,l'avantage principal est de viser à faciliter le processus de localisation des fautes par les programmeurs, en leur offrant une précision accrue et permettant ainsi une correction plus efficace des défauts de code où deux algorithmes ont été développés pour générer les motifs(ensembles de support) : l'algorithme glouton (heuristique) et l'algorithme exact (minimal hitting set - MHS en anglais). Ces motifs permettent d'identifier l'instruction fautive dans les programmes.

Pour tester et évaluer nos approches, nous avons choisi une base de données open source appelée ISSTA13 (International Symposium on Software Testing and Analysis) de l'année

2013 , cette base est une collection de tests de logiciels, de programmes et de résultats de recherche où contenant 10 programmes avec des fautes localisées à des emplacements différents .

Notre mémoire se compose de trois chapitres, accompagnés d'une introduction et d'une conclusion générale. Dans le premier chapitre, nous exposons les notions fondamentales de la localisation des fautes dans les programmes, tout en fournissant un état de l'art sur quelques approches existantes dans ce domaine.

Dans le deuxième chapitre, nous explorons l'Analyse Logique des Données (ALD) où nous examinerons plusieurs algorithmes de génération de motifs.

Le troisième chapitre est consacré à nos approches proposées où nous présentons en détail les différentes méthodes et algorithmes adoptés , nous présenterons également les résultats obtenus qui calculent via trois mesures de performances, à savoir l'exam score dans ses versions pessimistes et optimistes, ainsi que le delta Exam score . Nous terminons ce chapitre par la présentation de l'application développée.

Chapitre 1

Localisation de fautes dans les programmes

1.1 Introduction

“ La programmation est facile ; le débogage est difficile ; la localisation des fautes est encore plus difficile. ”
– Edsger Dijkstra –

Le développement de logiciels est un processus complexe et les logiciels sont sujets aux erreurs, appelées bugs. Ces erreurs peuvent avoir des conséquences néfastes sur la fiabilité et la qualité des logiciels, d'où l'importance de les détecter et de les corriger.

Le test logiciel est la méthode la plus utilisée pour valider les programmes et identifier les bugs. Il consiste à exécuter le logiciel avec des données spécifiques et à observer son comportement. Si le comportement observé ne correspond pas aux exigences et spécifications du logiciel, une défaillance est détectée, cette première étape est appelée étape de détection.

La localisation des fautes est la deuxième étape du processus de test logiciel. Elle vise à identifier la partie du système responsable de la défaillance. Cette étape est cruciale car elle permet de concentrer les efforts de correction sur une zone précise du code, ce qui réduit le temps et les ressources nécessaires pour corriger le bug dans troisième étape.

Il existe diverses techniques de localisation de fautes dans les programmes, chacune présentant ses propres avantages et inconvénients, que nous examinerons au fur et à mesure dans ce chapitre.

La localisation des fautes est un domaine de recherche actif et de nouvelles techniques sont constamment développées. L'objectif est de trouver des techniques plus précises, plus efficaces et moins coûteuses pour localiser les défauts dans les logiciels.

Dans ce chapitre nous allons parcourir quelques notions de base de cet univers complexe. En premier lieu, nous allons voir quelques notions de base sur la localisation des fautes et clarifierons son objectif. Ensuite, nous passerons en revue quelques techniques et approches existantes. Enfin, nous conclurons en synthétisant les points clés abordés.

1.2 Notions de base

1.2.1 Localisation de Fautes (LF)

Une fois que le programme ou le logiciel est en phase de diffusion, il serait contraignant d'avoir des retours en matière de debugging concernant la fonctionnalité du code, ce qui induit à une phase de maintenance. Afin d'éviter cela, le programmeur doit vérifier l'existence d'éventuelles fautes dans son code en faisant plusieurs tests. Il sera question de voir si le résultat est conforme ou pas, et par conséquent trouver l'origine du problème, Il s'agit alors de localiser la faute.

La philosophie des **fautes** est composée de trois notions principales :

- **Défaillance**: Différence entre le résultat attendu et le résultat obtenu d'un programme.
- **Erreur**: Partie du système susceptible de conduire à une défaillance.
- **Faute**: Cause supposée ou adjugée d'une erreur [Laprie, 1992].

Il existe une relation entre la faute, l'erreur et la défaillance, (voir la figure 1.1)[Avizienis et al., 2004] Lorsqu'une faute est activée (lorsque le programme l'exécute) elle génère une erreur,cette erreur se propage et engendre une défaillance dans le comportement du programme. Paradoxalement, une défaillance peut également être à l'origine d'une faute. Par exemple, si une méthode produit un résultat erroné et qu'elle est appelée ailleurs dans le programme, cette invocation devient une faute pour la suite de l'exécution.

Figure 1.1: Chaîne causale de menace a la fiabilité d'un logiciel

1.2.2 Les Tests

Pour comprendre le fonctionnement d'un système ou d'un programme dans différentes situations, on le lance avec des données d'entrée spécifiques. On appelle ces données des "cas de test". Dans ce contexte, nous aborderons ci-dessous quelques définitions fondamentales

1.2.2.1 Cas de test (on anglais test case (tc))

Un cas de test est caractérisé par deux principaux éléments:

D_i : Un ensemble de paramètres d'entrée qui permettent de déterminer si un programme P se comporte comme prévu.

O_i : La sortie attendue du programme pour l'ensemble D_i

1.2.2.2 Succès et échec d'un cas de test

Si la sortie courante A_i du programme P après l'exécution de D_i est égale à la sortie attendue O_i, le cas de test est considéré comme correct (positif), sinon, le cas de test est considéré comme erroné (négatif)

1.2.2.3 Suite de test

Une suite de test est un ensemble de **n** cas de test T = tc_1,tc_2 ,...,tc_n qui vise à tester si le programme P respecte l'ensemble des exigences spécifiées

1.2.2.4 Couverture de cas de test

Étant donné un cas de test tc_i et un programme P, la couverture du cas de test I_i est l'ensemble des instructions de P qui sont exécutées (au moins une fois) avec tc_i

- I_i = ($I_{i,1}$, ...,$I_{i,n}$) est un vecteur de couverture où $I_{i,j}$= 1 si l'instruction à la ligne j est exécutée et 0 sinon.
- La couverture d'un cas de test indique quelles parties du programme sont actives lors d'une exécution spécifique.

Par exemple, le cas de test tc_4 illustré dans la table 1.1 [Maamar et al., 2017] couvre les instructions (e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}). Son vecteur de couverture est I_4 = (1, 1, 1, 1, 0, 1, 1, 0, 0, 1).

Table 1.1: Exemple d'un programme avec la matrice de couverture associee

	programme: compteur de caractères	tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8

	function count (char *s){
	int let, dig, other, i = 0;
	char c;								

e1	: while (c = s[i++]) {	1	1	1	1	1	1	1	1
e2	: if('A'<=c && 'Z'>=c)	1	1	1	1	1	1	0	1
e3	: let += 2; //- faute -	1	1	1	1	1	1	0	0
e4	: else if ('a'<=c && 'z'>=c)	1	1	1	1	1	0	0	1
e5	: let += 1;	1	1	0	0	1	0	0	0
e6	: else if ('0'<=c && '9'>=c)	1	1	1	1	0	0	0	1
e7	: dig += 1;	0	1	0	1	0	0	0	0
e8	: else if (isprint (c))	1	0	1	0	0	0	0	1
e9	: other += 1;	1	0	1	0	0	0	0	1
e10	: printf("%d %d %d \n " , let, dig, other); }	1	1	1	1	1	1	1	1
	positif et négatif	N	N	N	N	N	N	P	P

1.2.3 Hypothèses fondamentales de la localisation de défauts

La plupart des approches de localisation de défauts reposent sur deux hypothèses clés:

1.2.3.1 L'hypothèse du développeur compétent

Cette hypothèse stipule que même si un programme contient plusieurs fautes simples [DeMillo et al., 1978], il sera capable de répondre à la quasi-totalité de ses spécifications, En d'autres termes, le programme fonctionnera plus ou moins correctement, malgré les erreurs présentes.

1.2.3.2 L'hypothèse de la faute simple

Cette hypothèse suppose que le programme ne contient qu'une seule instruction fautive. Cette exigence peut sembler restrictive, mais il a été démontré que les fautes complexes sont généralement le résultat d'une combinaison de fautes simples [Jones et al., 2002].

1.2.4 Les étapes de la localisation des fautes

La localisation des fautes est un processus crucial pour identifier les causes des échecs de programme. Elle peut être divisée en deux étapes principales :

1.2.4.1 Identification du code suspect

Diverses méthodes permettent d'analyser les cas de test et la couverture de code pour identifier les sections de code qui sont les plus susceptibles de contenir des bugs. Ces méthodes attribuent une priorité à chaque section de code en fonction de sa probabilité de contenir un défaut. Les sections de code avec une priorité plus élevée sont examinées en premier, car elles sont plus suspectes.

1.2.4.2 Vérification par le programmeur

Une fois que le code suspect a été identifié, il est examiné par un programmeur pour confirmer la présence d'un bug, s'il est trouvé, il est ensuite corrigé. L'efficacité de la localisation des défauts dépend de la précision des méthodes utilisées pour identifier le code suspect et de la capacité des programmeurs à détecter les bugs dans le code examiné.

voici un exemple de programme P comportant une faute et composé de n lignes L =e_1, e_2, ..., e_n. La table 1.1 illustre un exemple simple avec un programme nommé "Compteur de caractères". Ce programme contient 10 lignes (L =e_1, e_2, ..., e_{10}) et une faute à la ligne 3. L'instruction correcte à cette ligne devrait être **"let += 1;"**. Or, l'instruction erronée cause une erreur lors de l'exécution du programme

1.2.5 Localisation de fautes multiples dans les programmes

La plupart des méthodes de localisation de fautes supposent un seul défaut dans le programme. En pratique, les programmes peuvent contenir plusieurs fautes, ce qui représente un défi pour les méthodes classiques. Il existe plusieurs techniques pour traiter des programmes à fautes multiples telles que :

1.2.5.1 Focalisation sur les fautes

L'approche de focalisation sur les fautes vise à identifier et corriger plusieurs erreurs dans un programme. Cette technique fonctionne en regroupant les cas de test négatifs (ceux qui échouent) en clusters [Podgurski et al., 2003]

[Liu and Han, 2006][Jones et al., 2007][Wong et al., 2010]en fonction des fautes qu'ils exposent.

Cependant, pour utiliser cette technique efficacement, il est nécessaire de connaître le nombre de fautes présentes dans le programme à l'avance. Ce n'est pas toujours possible en pratique, car le nombre de fautes peut être difficile à déterminer.

Une alternative à la connaissance du nombre de fautes consiste à regrouper les cas de test négatifs par similarité des traces d'exécution. La trace d'exécution est un enregistrement des instructions du programme qui ont été exécutées lors d'un test.

En regroupant les cas de test avec des traces d'exécution similaires, il est possible de déduire qu'ils exposent probablement la même faute.

Une fois les cas de test regroupés en clusters, chaque cluster est ensuite combiné avec des cas de test positifs (ceux qui réussissent) pour identifier la faute correspondante. Les cas de test positifs permettent de déterminer quelle instruction du programme est à l'origine de l'erreur.

La focalisation sur les fautes est une technique efficace pour la localisation de fautes multiples dans les programmes; cependant, elle nécessite de connaître le nombre de fautes à l'avance ou d'utiliser des techniques de regroupement basées sur les traces d'exécution

1.2.5.2 Stratégie "une faute à la fois"

La stratégie "une faute à la fois" est une technique simple et intuitive pour la localisation de fautes multiples dans les programmes.[Wong and Qi, 2009].

Cette technique fonctionne en localisant et en corrigeant une seule faute à la fois, puis en exécutant le programme avec les mêmes cas de test pour identifier d'autres fautes potentielles. Ce processus est répété jusqu'à ce qu'il n'y ait plus de défaillance

1.3 Quelques approches de localisation de fautes

1.3.1 Approches traditionnelles

Les approches traditionnelles de localisation de fautes visent à identifier les erreurs dans un programme lorsqu'il présente un comportement anormal. Elles incluent :

1.3.1.1 Analyse de la mémoire et insertion de messages d'impression

L'analyse de la mémoire et l'insertion de messages d'impression sont deux approches couramment utilisées pour localiser les bugs dans un programme présentant un comportement anormal.[Wong et al., 2016]. La première méthode implique l'analyse d'un vidage de mémoire correspondant à l'exécution du programme. Bien que cette méthode soit encore utilisée aujourd'hui, elle nécessite l'analyse d'un volume considérable de données, ce qui peut être fastidieux. Une alternative à cette approche consiste à insérer des messages

d'impression autour du code suspect. Cette technique permet d'afficher la valeur des variables clés, facilitant ainsi la détection des erreurs. Cependant, elle requiert une compréhension approfondie de l'exécution du programme par rapport au cas de test défaillant. De plus, il est crucial d'insérer la quantité exacte de messages aux endroits appropriés, ce qui peut s'avérer complexe

1.3.1.2 Utilisation d'outils de debug

Pour pallier ces limitations, des outils de debug comme DBX ou le débogueur Microsoft VC++ ont été développés. Ils permettent de définir des points d'arrêt stratégiques et d'examiner à ces points l'état des variables et du programme. On peut considérer cela comme l'insertion virtuelle de messages d'impression, sans modification du code. Ces points d'arrêt peuvent être définis avant l'exécution ou dynamiquement pendant celle-ci. L'exécution peut se faire en continu d'un point d'arrêt à l'autre, ou par pas à partir d'un point d'arrêt.

L'utilisateur doit d'abord définir ces points stratégiques. À chaque point d'arrêt, il peut analyser l'état du programme et déterminer si l'exécution est toujours correcte. Si ce n'est pas le cas, l'exécution est interrompue et une analyse plus approfondie peut être nécessaire pour localiser le bug. Sinon, l'exécution se poursuivre en mode continu ou par pas.

Ces outils fournissent un instantané de l'état du programme à différents points d'une exécution

1.3.2 Approches avancées

1.3.2.1 Approches basée sur le découpage de programme

Le découpage de programme est une technique courante pour le débogage [Tip, 1994]. Il permet d'identifier des sections de code susceptibles de contenir des bugs en se basant sur une variable et sa valeur à un certain point du programme. Il existe plusieurs types de découpage, à savoir :

1. **Découpage statique (en anglais static program slice)**

   analyse le code source et identifie toutes les instructions pouvant influencer la valeur d'une variable donnée à un point donné. Cette approche peut inclure des instructions qui ne seront jamais exécutées en réalité.

   Dans ce contexte la,[Dutta, 2024] ont procéder à une localisation des fautes dans un programme défectueux sans assertions spécifiées par l'utilisateur et sans exécution des programmes, et donc sans utiliser de cas de test, La méthode proposée réduit l'espace de recherche de défauts en supprimant les régions équivalentes du code produit à l'aide de techniques de vérification.

2. **Découpage dynamique (en anglais dynamic program slice)** analyse l'exécution du programme et identifie uniquement les instructions ayant effectivement affecté la valeur d'une variable. Cette méthode est plus précise mais peut être plus lourde à calculer.

3. **Tranche d'exécution (en anglais execution slice)** analyse l'exécution d'un test spécifique et identifie les instructions exécutées par ce test. Cette approche est plus efficace que le découpage statique car elle se base sur un cas d'exécution concret.

Les méthodes de découpage peuvent réduire la zone de recherche des bugs en se concentrant sur des sections de code pertinentes. Cependant, il est possible que le bug se situe en dehors de la zone identifiée. Des techniques supplémentaires peuvent être utilisées pour affiner la recherche

1.3.2.2 Approche basées sur le spectre du programme (en anglais Spectrum-Based Fault localization (SBFL))

Les méthodes basées sur le spectre enregistrent des informations sur l'exécution d'un programme, telles que les branches conditionnelles ou les boucles empruntées. En cas d'échec, ces informations peuvent ensuite être utilisées pour identifier les parties du programme les plus susceptibles de contenir des bugs. Les premières études considéraient seulement les cas de test négatifs, par la suite une considération des deux cas de test positif et négatif à permit d'aboutir à de meilleurs résultats [Wong et al., 2016].

Plusieurs travaux ont été réalisés en considérant l'information liée aux exécutions positives/négatives tels que Renieris et Reiss [Renieres and Reiss, 2003] qui proposent l'utilisation de l'Union et l'Intersection entre ces deux typez d'exécutions. Leur méthode se concentre sur le code qui s'exécute uniquement pendant les tests échoués (négatifs), et qui est donc plus susceptible de contenir des bugs (fautes).

D'autres auteurs ont fait recours aux mesures de suspicion lors de la détection de la faute. Il s'agit d'attribuer un score de suspicion à chaque portion de code, puis classer ces scores. Les instructions avec le score le plus élevé sont les plus suspectes. Toutes ces mesures traduisent le fait que les instructions là sont celles exécutées le plus souvent par les cas de tests négatifs et le moins souvent par les cas de tests positifs, ce qui traduit une corrélation entre les cas de tests négatifs et l'instruction fautive

Plusieurs mesures de classement ont été proposées pour capturer la notion de méfiance, telles que **Tarantula** [Jones and Harrold, 2005],**Ochiai** [Abreu et al., 2007] et **Jaccard**[Abreu et al., 2007].
L'objectif de SBFL est de disposer d'une métrique capable de toujours classer en premier les énoncés erronés. En pratique, nous sommes très loin de cet idéal . Les métriques SBFL ne reposent pas sur un modèle particulier du programme testé et sont donc faciles à utiliser et pratiques en présence de contraintes de temps CPU et de ressources mémoire.

Les métriques SBFL donnent différentes interprétations du degré de suspicion. De plus, la sémantique des instructions et les dépendances ne sont pas prises en compte. Ainsi, la précision des approches SBFL est intrinsèquement limitée.

1.3.2.3 Méthodes basées sur des statistiques

Les méthodes statistiques permettent d'identifier des zones suspectes dans le code source susceptibles de contenir des bugs. Dans ce contexte Liblit et al.[Liblit et al., 2005] propose un algorithme nommé **Liblit05** qui analyse des informations générées par des instructions d'un programme. Il calcule la probabilité qu'une instruction soit vraie et implique un échec du programme. Les instructions avec un score élevé sont examinées en priorité par les programmeurs pour trouver les bugs.

Wong et al. [Wong et al., 2008] utilise une analyse par tableaux classification croisée appelée **Crosstab** (en anglais cross-classification table) pour calculer le niveau de suspicion de chaque instruction. Cette méthode classe toutes les instructions du programme en fonction de leur probabilité de contenir des bugs, contrairement à Liblit05 qui se focalisent sur des instructions spécifiques.

Une extension d'une étude récente [Wong et al., 2008] rapporte que Crosstab est presque toujours plus efficace pour localiser les bugs dans la suite Siemens que Liblit05

1.3.2.4 Méthodes basée sur l'état du programme

L'état d'un programme correspond aux valeurs de ses variables à un moment donné de son exécution. Cet état peut être un bon indicateur pour localiser les bugs. Une approche courante consiste à modifier les valeurs de variables afin d'identifier celles qui causent une exécution erronée.

Dans un travail antérieur Zeller et al. [Zeller, 2002] mis en évidence les limites de la localisation d'erreurs basée uniquement sur l'analyse de l'espace de recherche (variables, valeurs). Il montre qu'il ne suffit pas de comparer les états du programme dans les cas de test positifs et négatifs pour identifier les instructions fautives. En effet, une erreur peut avoir des répercussions sur tous les états suivants du programme,rendant difficile la localisation précise de la source du problème.

Cleve et Zeller [Cleve and Zeller, 2005] proposent une méthode de transition de cause qui identifie le moment et l'emplacement où la cause de l'échec change d'une variable à une autre. Cette méthode peut être coûteuse car elle nécessite de nombreux tests supplémentaires. Alors que Gupta et son équipe [Gupta et al., 2005] développent une extension de la méthode de transition de cause qui utilise des "tranches" dynamiques pour identifier le code suspect.

Zhang et al.[Zhang et al., 2006] modifient les états du programme (basculement de

prédicat) pour forcer l'exécution de branches différentes dans un test échoué. Un prédicat dont la modification permet une exécution réussie est considéré comme critique et potentiellement responsable du bug.

Wang et Roychoudhury [Wang and Roychoudhury, 2005] analyse automatiquement le chemin d'exécution d'un test échoué et modifie le résultat des branches pour aboutir à une exécution réussie. Les branches modifiées sont considérées comme des bugs.

Ces techniques à base d'état de programme permettent d'identifier des variables et des sections de code suspectes, facilitant ainsi la localisation des fautes logicielles.

1.3.2.5 Méthodes basées sur l'Apprentissage Automatique(AA)

L'apprentissage automatique (ou le Machine Learning en anglais(ML)) est un domaine de l'informatique qui étudie les algorithmes capables de s'améliorer automatiquement grâce à l'expérience. Ces techniques, adaptatives et robustes, permettent de construire des modèles à partir de données avec une intervention humaine limitée. L'apprentissage automatique est utilisé dans de nombreux domaines tels que le traitement automatique du langage, la cryptographie, la bio-informatique, la vision par ordinateur, etc.

Dans le contexte du diagnostic logiciel, on peut voir la localisation d'une erreur comme un apprentissage à partir de données. L'objectif est d'apprendre ou de déduire l'emplacement d'un bug en se basant sur des informations telles que la couverture de code par les tests. Il n'est donc pas surprenant que de nombreux chercheurs se soient intéressés à l'application des techniques d'apprentissage automatique pour la localisation des fautes logicielles.

Parmi ces chercheurs, Wong et Qi [Wong and Qi, 2009] qui ont fait recours aux Réseaux de Neurones Artificiel(RNA) afin d'apprendre des relations complexes entre les données de couverture du code (quelles parties du code sont utilisées par les tests) et les résultats des tests (succès ou échec). En analysant cette relation, le réseau peut identifier les sections de code plus susceptibles de contenir des fautes.

Les arbres de décision (AD) [Briand et al., 2007] ont aussi étés utilisés dans ce contexte. Ces algorithmes créent une série de règles pour classer les données. Dans la localisation de fautes, ils peuvent être utilisés pour grouper les cas de test en fonction de leur comportement et identifier les sections de code potentiellement responsables des défaillances dans chaque groupe.

Brun et Ernst [Brun and Ernst, 2004] construisent un modèle d'apprentissage utilisant les Machines à Vecteurs de Support (MVS) et les arbres de décision pour trouver automatiquement des erreurs dans les programmes. Pour cela, ils analysent le code lui-même (analyse statique) et recherchent des motifs indicateurs d'erreurs. Par exemple, une variable sans valeur initiale peut être un indice.

Le modèle apprend en comparant ces motifs à partir de programmes fonctionnels et défaillants. Ainsi, lorsqu'on lui présente un nouveau programme, il peut classer les différentes

parties du code en fonction de leur probabilité de causer des problèmes. Cela aide les programmeurs à identifier beaucoup plus rapidement les zones où chercher les bugs.

[Wang et al., 2022]proposent une approche basée sur un modèle d'apprentissage profond de type "Wide and Deep" pour la localisation des failles logicielles. Cette approche se distingue par son exploitation des interactions entre les différentes caractéristiques du code permettant d'évaluer la suspicion de présence d'une erreur, telles que le comportement des appels de fonctions, les invariants du programme, les mesures statiques du code, etc. Le modèle "Wide et Deep" combine un apprentissage linéaire ("Wide") pour les relations simples et un apprentissage non linéaire profond ("Deep") pour les interactions complexes. Cela permet de capturer plus efficacement les indicateurs de présence d'une erreur. Les résultats expérimentaux sur un jeu de données standard de défauts logiciels montrent que cette approche surpasse les méthodes traditionnelles et les approches d'apprentissage profond de pointe en termes de précision et de détection précoce des failles.

En conclusion, l'apprentissage automatique offre une voie prometteuse pour la localisation automatisée des fautes logicielles, offrant diverses techniques pour analyser le comportement du programme et identifier la cause principale des problèmes

1.3.2.6 Méthodes basées sur le Data Mining

La fouille de données de données (ou Data Mining en anglais (DM)) vise à créer un modèle ou une règle à partir d'informations pertinentes extraites des données. Il permet de découvrir des patterns cachés dans des ensembles de données volumineux, souvent impossibles à identifier par une analyse manuelle. L'objectif est d'identifier le motif d'exécution d'instructions menant à un échec du programme.

Les traces d'exécution d'un programme (enregistrement des instructions exécutées lors d'un test) sont une ressource précieuse pour localiser les fautes. Cependant, leur volume important rend leur analyse manuelle difficile. Des études ont appliqué avec succès des techniques de Data Mining pour traiter ces traces volumineuses.

Nessa et al.[Nessa et al., 2008] proposent une méthode de localisation de bug basée sur l'analyse des traces d'exécution. Ils découpent les traces d'exécution en séquences d'instructions appelées N-grammes et analysent les N-grammes présents plus fréquemment dans les exécutions échouées. Pour chaque N-gramme identifié, ils calculent la probabilité qu'une exécution échoue si ce N-gramme apparaît dans sa trace, ce qu'ils appellent la "confiance". Ils classent ensuite les N-grammes par ordre décroissant de confiance et remontent au code source pour identifier les instructions correspondantes, en commençant par celles ayant la plus haute confiance.

Cellier et al.[Cellier et al., 2008] proposent une méthode utilisant des règles d'association et l'Analyse de Concepts Formel (ACF) pour la localisation de fautes. Cette approche vise à identifier des règles entre l'exécution d'instructions et les échecs de tests associés, puis à mesurer la fréquence de chaque règle. Un seuil est ensuite défini pour sélectionner les règles couvrant un nombre minimum d'échecs.Ces nombreuses règles peuvent être

partiellement ordonnées à l'aide d'un treillis de règles, puis explorées de manière ascendante pour détecter la faute

1.4 Conclusion

Dans ce chapitre, nous avons abordé quelques notions fondamentales de la localisation des fautes ainsi que plusieurs approches utilisées à cette fin. Ces approches ont connu une évolution constante. Malheureusement, les logiciels deviennent de plus en plus complexes, ce qui signifie que les défis posés par la localisation des fautes augmentent également. Ainsi, il reste encore beaucoup de recherches à mener et de percées à réaliser pour améliorer l'efficacité et la précision des méthodes de localisation des fautes. Dans le chapitre suivant, nous examinerons une méthode issue de la fouille de données : l'Analyse Logique des Données (ALD), que nous avons précisément adoptée dans ce contexte.

Chapitre 2

Analyse logique des données

2.1 Introduction

L'analyse de données existe depuis des siècles, avec des exemples remontant aux recensements de bétail dans les premières structures sociales. L'empire romain a utilisé des recensements pour connaître ses ressources et classer ses citoyens.

Cependant, ce n'est qu'au milieu du XXe siècle que l'on a pris conscience du potentiel des grandes quantités de données. L'augmentation de la puissance de calcul a permis le développement de nombreuses méthodes d'extraction de connaissances à partir de ces données, appelées analyse de données ou apprentissage automatique.

Peter L. Hammer a proposé une alternative originale aux méthodes statistiques : l'Analyse Logique des Données (ALD, ou LAD en anglais pour Logical Analyse Data) [Hammer, 1986], une méthode d'exploration de données qui repose sur l'identification de patterns, des combinaisons spécifiques de valeurs binaires, qui permettent de discriminer avec précision des sous-ensembles au sein d'un groupe de données.

Contrairement aux méthodes statistiques traditionnelles, qui se focalisent sur la description et la prédiction des données, l'ALD permet la détermination des causes profondes qui sous-tendent la formation des groupes de données. Cette approche permet de mettre en lumière les interactions et les relations entre les variables, offrant ainsi une compréhension plus fine et plus exploitable des données.

L'originalité de l'ALD réside dans sa capacité à:

- Détecter des structures et des relations cachées dans les données que les méthodes statistiques classiques pourraient ne pas saisir.
- Expliquer les raisons pour lesquelles certains groupes de données se distinguent des autres.
- Procurer des informations exploitables pour la prise de décision et l'optimisation des processus.

L'ALD s'avère particulièrement utile dans des domaines variés tels que l'informatique, la recherche scientifique , la médecine [Valizadeh et al., 2024], la finance et le marketing. C'est un domaine en pleine expansion et de nombreux développements méthodologiques et applicatifs sont en cours. L'avenir de l'ALD est prometteur et son potentiel pour l'exploration et l'exploitation des données est immense.

Dans ce chapitre nous allons d'abord donner une petite présentation de la méthode de l'analyse logique de données ALD, Nous verrons par la suite la méthode de **"binarisation"** indispensable pour les données a valeur continu, puis nous exposerons quelques techniques d'extraction de motif .Nous clôturons ce chapitre par une conclusion.

2.2 Présentation de ALD

L'analyse logique de données introduite pour la première fois par Peter Hammer [Hammer, 1986] qui est une méthode d'analyse de données utilisant l'optimisation combinatoire et les fonctions booléennes partiellement définies. Elle vise à identifier des motifs (patterns) dans les données, c'est-à-dire des ensembles de valeurs communes à un groupe d'observations.

L'objectif est de caractériser les données et de justifier explicitement les groupes de données, contrairement aux approches de classification traditionnelles qui se concentrent sur la construction de groupes.

L'analyse logique de données permet de :

- Déterminer des similarités entre des individus d'un même groupe.
- Discriminer les individus n'appartenant pas à un groupe.
- Déduire des règles logiques propres à chaque groupe.

La figure 2.1 utilise une zone hachurée pour représenter un motif d'un ensemble de points noirs. Cette zone ne recouvre que des points noirs, soulignant ainsi leur similarité et leur proximité.

Ce motif ne recouvre pas nécessairement tous les points d'un groupe. Dans l'exemple, agrandir la zone pour inclure les deux points noirs restants inclurait également un point blanc, ce qui n'est pas acceptable.

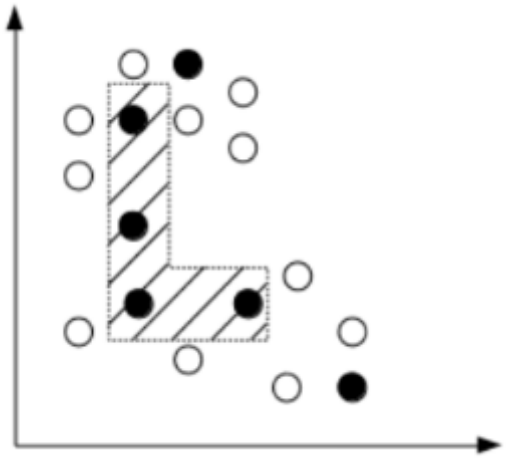

Figure 2.1: représentation naïve d'un pattern

Pour comprendre le concept, nous allons prendre l'exemple d'un joueur d'échecs qui étudie 7 parties (numérotées de 1 à 7) et tente de déterminer l'impact de ses choix

stratégiques. Il a utilisé 8 stratégies différentes (numérotées de a à h) et a perdu 3 parties sur les 7.[Chambon, 2017]

La table ci-dessous résume ses résultats, sachant que:

- $X_{i,j} = 1$ si la stratégie j a été utilisée dans la partie i.
- $X_{i,j} = 0$ si la stratégie j n'a pas été utilisée dans la partie i.

Table 2.1: Résultats des 7 parties de l'exemple du joueur d'échec

Parties	Résultat	Stratégies							
		a	b	c	d	e	f	g	h
1	Défaite	0	1	0	1	0	1	1	0
2		1	1	0	1	1	0	0	1
3		0	1	1	0	1	0	0	1
4	Victoire	1	0	1	0	1	0	1	1
5		0	0	0	1	1	1	0	0
6		1	1	0	1	0	1	0	1
7		0	0	1	0	1	0	1	0

Un examen rapide du table révèle des observations intéressantes:

- La stratégie **b** n'est jamais utilisée sans la stratégie **a** dans les parties gagnées, mais cela arrive dans les parties perdues (De là nous concluons que **b** sans **a** semble être un motif des parties perdues).
- Dans les parties gagnées, **f** ou **g** est toujours utilisée, tandis que dans les parties perdues, les deux sont utilisées ou aucune (De là nous concluons que **f** et **g** semble être un **motif** des parties perdues ainsi que n'utiliser ni **f** et ni **g**)

Ces observations permettent d'identifier des patterns qui peuvent expliquer les résultats des parties.

2.3 Binarisation

L'analyse logique des données (ALD) a été initialement développée pour analyser des ensembles de données dont les attributs ne prennent que des valeurs binaires (0-1). Cependant, la plupart des applications réelles impliquent des attributs à valeurs continues. Pour traiter ces données, une méthode de binarisation est appliquée [Boros et al., 1997]. Elle consiste à associer un ensemble d'attributs binaires à chaque attribut numérique. Chaque attribut binaire prend la valeur 1 si la valeur de l'attribut numérique associé est supérieure à un certain seuil, et 0 sinon. Le défi consiste à trouver le nombre minimal de seuils (points de coupure) permettant de distinguer les observations positives et des observations négatives tout en préservant l'intégrité des données.

Le processus de trouver le nombre minimum de points de coupure est un problème informatique **complexe**. Pour pallier cela, la question alors de réduire le nombre de points de coupure à analyser en ordonnant d'abord les données en fonction d'un attribut spécifique , et à ne considérer par la suite que les seuils situés entre des valeurs où le signe des données change (de positif à négatif ou vice versa).

Même après cette simplification, il peut encore rester des millions de points de coupure potentiels à examiner, des techniques simples sont utilisées pour éliminer les points de coupure non pertinents avant d'appliquer les techniques ALD standard. Ces techniques consistent notamment à trouver des corrélations entre les points de coupure et les résultats, et à éliminer les points de coupure qui ne permettent pas de distinguer un grand nombre d'observations positives et négatives.

2.4 Génération des motifs

Cette section s'intéresse à la génération de patterns, une étape cruciale dans l'exploration des données binaires. Diverses techniques existent permettant la découverte de patterns. Nous allons voir dans ce qui suit quelques unes.

2.4.1 Génération des motifs primaires (primes patterns)

Les motifs primaires sont des schémas logiques fondamentaux qui peuvent être des structures récurrentes ou des modèles qui se produisent fréquemment dans les données analysées. Ils constituent une base essentielle pour l'exploration ultérieure. L'algorithme de [Boros et al., 2000] permet à la génération exhaustive de ces motifs dit primaires à partir d'un ensemble de données binaires. Son principe repose sur l'exploration systématique de tous les motifs jusqu'à un degré maximal .

L'algorithme de Boros garantit la génération de tous les motifs primaires car il explore systématiquement toutes les combinaisons possibles de littéraux.

2.4.2 Génération des motifs étendus (spanned patterns)

Un autre algorithme est proposé pour identifier l'ensemble des "motifs étendus". Cette catégorie de patterns offre une couverture plus large des données, tout en conservant une certaine structure.

L'algorithme **SPIC** (Spanned Patterns via Input Consensus), développé par [Alexe and Hammer, 2006], s'attaque à la génération de "motifs étendus". Pour ce faire, il introduit une nouvelle notion d'intervalle appelée **"consensus"**. Prenons deux vecteurs booléens α et β de **n** variables. L'intervalle généré par ces vecteurs est un couple de vecteurs booléens [α', β'] tel que pour chaque i entre 1 et n , $\alpha'_i = min(\alpha_i, \beta_i)$ et $\beta'_i = max(\alpha_i, \beta_i)$.

Par exemple, si α= (0, 1, 1) et β = (1, 0, 1), l'intervalle généré sera [(0, 0, 1), (1, 1,1)].

Un vecteur booléen **y** sera dit **"couvert"** par l'intervalle [α', β'] si pour chaque i entre 1 et n, $\alpha'_i \leq y_i \leq \beta'_i$.

Par l'exemple,le vecteur y = (1, 1, 1) est couvert par [α', β'] tandis que
$y' = (1, 1, 0)$ ne l'est pas.

À partir d'un intervalle [α, β] défini par deux vecteurs booléens de **n** variables. Un terme **t** peut être construit à partir de cet intervalle comme suit:

$$t = \left(\bigwedge_{i \text{ t.q. } \alpha_i=\beta_i=1} l_i \right) \wedge \left(\bigwedge_{j \text{ t.q. } \alpha_j=\beta_j=0} \overline{l_j} \right) \tag{2.1}$$

Où l_i représente la i-ème variable du jeu de données.

De manière symétrique, un intervalle [α, β] peut être reconstruit à partir d'un terme **t**.

Pour chaque i entre 1 et n:

- $\alpha_i = 1$ si $l_i \in Lit(t)$ (l'ensemble des littéraux de t) et 0 sinon.
- $\beta_i = 0$ si $\overline{l_i} \in Lit(t)$ et 1 sinon.

Il existe donc une équivalence bijective entre un terme et son intervalle correspondant. Cette équivalence implique que la notion de couverture d'un intervalle est identique à la notion de couverture d'un terme. un terme (ou son intervalle équivalent) définit un **pattern** s'il satisfait deux conditions:

- Il couvre au moins une observation positive du jeu de données.
- Il ne couvre aucune observation négative du jeu de données.

Reprenons l'exemple avec les vecteurs α et β avec α= (0, 1, 1) et β = (1, 0, 1) et associant les variables $l_1 = a$,$l_2 = b$, $l_3 = c$ .Le terme généré par l'intervalle [α, β] est t = c et t couvre l'observation $y = (1, 1, 1)$ car c est présent dans l'observation et les autres variables ne sont pas contraires et ne couvre pas l'observation y_0 = (1, 1, 0) car c n'est pas présent dans l'observation.

Le consensus entre deux patterns générés par les intervalles [α, β] et [α', β'] est un nouvel intervalle [α", β"] défini comme suit :

Pour chaque variable i entre 1 et n : $\alpha"_i = min(\alpha_i, \alpha'_i)$ et $\beta_{]i} = max(\beta_i, \beta'_i)$.

Le consensus étant un intervalle, il peut être associé à un terme, qui lui-même peut être un pattern. Cela permet d'exploiter le consensus pour générer de nouveaux patterns ou pour simplifier des patterns existants.

Le consensus entre deux patterns P_1 et p_2 peut être défini comme un terme **t** tel que :
Lit(t) = Lit(P_1) ∩ Lit(p_2).
En d'autres termes, le consensus ne conserve que les littéraux présents dans les deux patterns.
Notons que si p_2 est déjà un consensus entre P_1 et un troisième pattern, le consensus entre P_1 et p_2 sera identique à p_2.
Cela garantit la cohérence et la stabilité du processus de génération de consensus.
Notons également qu'un pattern P_1 réalisé par un consensus entre deux patterns p_2 et p_3 sera nécessairement de degré inférieur à p_2 et p_3, sauf si P_1 est identique à l'un des deux.
Cette propriété est importante pour la recherche de patterns efficients et précis.Pour chaque paire de motifs étendus (P_1, p_2), l'algorithme calcule un consensus.
Ce consensus est un nouveau pattern qui couvre l'ensemble des observations couvertes par P_1 et p_2,
Cov(p_1) ∪ Cov(p_2),tout en minimisant le nombre de littéraux retirés.

En explorant tous les consensus possibles, l'algorithme SPIC est capable de déterminer tous les motifs étendus couvrant chaque combinaison d'observations possible. L'ensemble initial de motifs étendus de départ correspond simplement aux patterns de degré n couvrant chaque observation positive individuellement.

Nous présentons l'algorithme SPIC (Algorithme 1) ci-dessous puis le décrirons en suivant un exemple.

Données : C_0 : l'ensamble spanned patterns couvrant chaque observation positive une a une
Résultat : C : l'ensemble des spanned patterns.

```
1  i = 0, C = C_0, W_0 = C_0;
2  tant que W_i = ∅ faire
3  |  W_{i+1} = ∅ ;
4  |  pour tous les p_i ∈ C_0 et p_2 ∈ W_i faire
5  |  |  p' = le concensus de p_1 et p_2;
6  |  |  si p' est un pattern et p' ∉ C alors
7  |  |  |  C = C ∪ {p'} ;
8  |  |  |  W_{i+1} = W_{i+1} ∪ {p'} ;
9  |  |  fin
10 |  |  i++;
11 |  fin
12 |  retourner C
13 fin
```

Algorithme 1 : L'algorithme de Calcul des spanned patterns

Voici un exemple appliquant des motifs étendus (Algorithme 1)

Table 2.2: Observations classées par groupes

Observation	Groupes	Variables		
		a	b	c
1	P	1	0	0
2		0	1	0
3		0	1	1
4	N	1	1	1
5		1	1	0

Étape 1 : Initialisation

- C_0 est composé de 3 patterns de degré 3 couvrant les observations 1, 2 et 3 : $C_0 = P_1, P_2, P_3 = \{a \wedge \overline{b} \wedge \overline{c}, \overline{a} \wedge b \wedge \overline{c}, \overline{a} \wedge b \wedge c\}$.
- On transforme les patterns en intervalles :
  p_1 = [(1 0 0),(1 0 0)]
  p_2 = [(0 1 0),(0 1 0)]
  p_3 = [(0 1 1),(0 1 1)]
- On initialise C = C_0 et $W_0 = C_0$.
- On initialise i = 0.

Étape 2 : Itération Boucle i = 0:

- W1 = $\emptyset$.
- On calcule les consensus :
  $p_{1,2}$ = [(1 0 0),(0 1 0)] (correspond à $\overline{c}$, couvre l'observation 5, non un pattern).
  $p_{1,3}$ = [(1 0 0),(0 1 1)] (terme vide, non un pattern).
  $p_{2,3}$ = [(0 1 0),(0 1 1)] (correspond à $\overline{a} \wedge$ b, ne couvre aucune observation négative, un pattern).
- On met à jour C et W1 :
  C = C $\cup$ $P_{2,3}$
  $W_1 = W_1 \cup P_{2,3}$
- On incrémente i : i = 1.

Boucle i = 1:

- $W_2 = \emptyset$.
- Il n'y a qu'un seul consensus à calculer :
  $p_{1,2,3}$ = [(0 0 0),(1 1 1)] (identique à $p_{1,3}$, couvre toutes les observations, non un pattern).
- On incrémente i : i = 2.

Étape 3 : Résultat final

- $W_2 = \emptyset$, on sort de la boucle.
- L'ensemble final des motifs étendus est C = $\{p_1, p_2, p_3, p_{2,3}\}$:
  $p_1 = \text{a} \wedge \overline{b} \wedge \overline{c}$
  $p_2 = \overline{a} \wedge \text{b} \wedge \overline{c}$
  $p_3 = \overline{a} \wedge \text{b} \wedge \text{c}$
  $p_{2,3} = \overline{a} \wedge \text{b}$

2.5 Conclusion

Dans ce chapitre nous avons parcouru quelques notions de base de l'analyse logique de donnée dont l'objectif est la détection des motifs plus important (fréquent) ce qui peut être exploité dans le domaine de localisation des fautes dans les programmes où ces motifs peuvent effectivement être les instruction susceptible d'être suspectes (contiennent des fautes).

Dans le chapitre suivant, nous abordons nos approches proposées basées justement sur l'ALD afin de générer des motifs discriminant deux classes d'instruction, instructions exécutées avec succès et instructions échouées. Ces motifs-là peuvent contenir des fautes.

Chapitre 3

Approche proposée, résultats et présentation de l'application

3.1 Introduction

La localisation des fautes dans les programmes est une étape essentielle pour identifier la partie du système à l'origine de la panne, elle concentre les efforts de correction sur une zone précise du code, ce qui optimise le temps et les ressources alloués à la résolution du problème.

Dans ce chapitre nous présentons nos approches proposées dédiées à la localisation des fautes dans les programmes, durant lequel nous nous appuyons sur les définitions introduites dans les chapitres précédents.

Nous avons introduit dans ce chapitre deux algorithmes, l'algorithme glouton [Greenberg, 1998] et l'algorithme d'extraction des traverses minimales d'un hypergraphe nommé aussi méthode exacte (en anglais Minimal Hitting set - MHS) [Gainer-Dewar and Vera-Licona, 2017]. Nos approches sont basées sur l'analyse logique des données (ALD) pour générer des ensembles minimaux de support. Afin de sélectionner le meilleur support parmi ceux qui sont générés qui devrait contenir l'erreur, nous avons utilisé les mesures de suspicions à savoir Tarantula [Jones and Harrold, 2005] , GP13 [Yoo, 2012], Jaccard [Abreu et al., 2007], Ochiai [Abreu et al., 2007]. Nous avons également introduit deux nouvelles mesures de suspicion, le XOR et le N et !P.

L'efficacité de ces algorithmes est évaluée à l'aide d'une base de données open source appelée ISSTA13 ("International Symposium on Software Testing and Analysis",Year 2013)lien de la base.

L'efficacité de nos approches est évaluée via trois mesures de performances, à savoir l'exam score dans ses versions pessimistes et optimistes, ainsi que le delta Exam score. les résultats obtenu sont satisfaisant et démentre l'efficacité de nos approches.

3.2 Présentation de la base de données

Pour valider l'efficacité de nos approches d'ALD dans la localisation des fautes dans les programmes, nous avons utilisé une base de données open source appelée ISSTA13 ("International Symposium on Software Testing and Analysis",Year 2013)lien de la base. cette base est une collection de tests de logiciels, de programmes et de résultats de recherche où contient 10 programmes avec plusieurs versions chacun. Chaque programme comporte une seule faute localisée à un emplacement différent dans chaque version.

L'index de l'erreur est indiqué dans un fichier vide portant le même nom que le programme.

Chaque version du programme comprend deux fichiers : B^+ et B^-. Le fichier B^+ représente les exécutions réussies du programme, tandis que le fichier B^- correspond

aux exécutions échouées. Les lignes de ces fichiers représentent les différents tests effectués, et les colonnes indiquent les instructions du programme (attributs). La valeur 1 est attribuée à une instruction exécutée lors d'un test, tandis que la valeur 0 indique une instruction non exécutée.

La **Table 3.1** présente un extrait du fichier B^+ du programme Daikon version 33. La structure des colonnes est identique dans le fichier B^-, à l'exception du nombre de tests (lignes) qui diffèrent.

Le programme contient **1938** colonnes qui représentent des instructions, et 156 lignes qui représentent des tests.

Table 3.1: Extrait de fichier B^+ de programme Daikon de la base ISSTA13 lien de la base

B+	e1	e2	e3	e4	e5	e6	...	...	e1938
tc1	0	0	0	0	0	1	...	...	0
tc2	0	0	0	0	0	0	...	...	0
tc3	0	0	0	0	0	1	...	...	0
tc4	0	0	0	0	0	1	...	...	0
...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
tc156	0	0	0	0	0	0	...	...	0

3.3 Approches proposées

Nous avons implémenté deux approches, la première est basée sur l'algorithme de glouton et la seconde sur la méthode exacte. Nos approches de localisation de fautes s'appuient sur l'analyse logique des données. Elles se décomposent en trois étapes :

1. Calcul des sous-ensembles d'instructions suspectes ou susceptibles de circonscrire les fautes. Ce calcul se base sur l'analyse logique des données (ALD) dont l'objectif est de calculer le sous-ensemble d'attributs (ensembles support) qui discriminent la classe positive de la classe négative.

2. Tri des ensembles support du plus suspect au moins suspect. Deux mesures ont été proposées pour calculer un tel ordre, la mesure XOR et la mesure N et !P (définies dans la suite).

3. Évaluation de nos approches sur des bases de tests réelles en utilisant nos mesures et celles de l'état de l'art.

Nous donnons quelques définitions formelles des notions évoquées ci-dessus. Soit $F_B = \{x1, \ldots, xt\} \in \{0,1\}^t$ un ensemble de $\mathbf{t}$ attributs (features) binaires, une base

de données binaire $B = (B^+, B^-)$.
Étant donné un ensemble d'attributs binaires F_B , on définit $B^+(S)(resp.B^-(S))$ comme la projection de $B^+(S)(resp.B^-(S))$ dans S .
S est appelé un ensemble support si $B^+(S) \cap B^-(S) = \emptyset$. De plus, S est dit non redondant ou minimal si aucun de ses sous-ensembles n'est un ensemble support.

Les ensembles minimaux de support (minimal support sets) est un sur-ensemble des ensembles support de taille minimum (minimum support sets). Pour contourner la complexité élevée de ce problème d'énumération, nous proposons deux approches :

1. **Algorithme glouton** : Pour générer un ensemble minimal de support nous adoptons trois types d'algorithmes gloutons, glouton de haut en bas, glouton de bas en haut et glouton hybride (voir les algorithmes ci-dessous). La complexité de ces algorithmes gloutons est linéaire O(n) où n est le nombre d'attributs.

2. **Algorithme exacte** : Cet algorithme d'énumération des ensembles minimaux de supports s'appuie sur une transformation du problème à celui de l'énumération des hitting sets minimaux dans un hypergraphe (voir ci-dessous)

3.3.1 Organigramme de nos approches proposées

L'organigramme présenté par la figure 3.1 décrit les principales étapes de nos approches proposées.

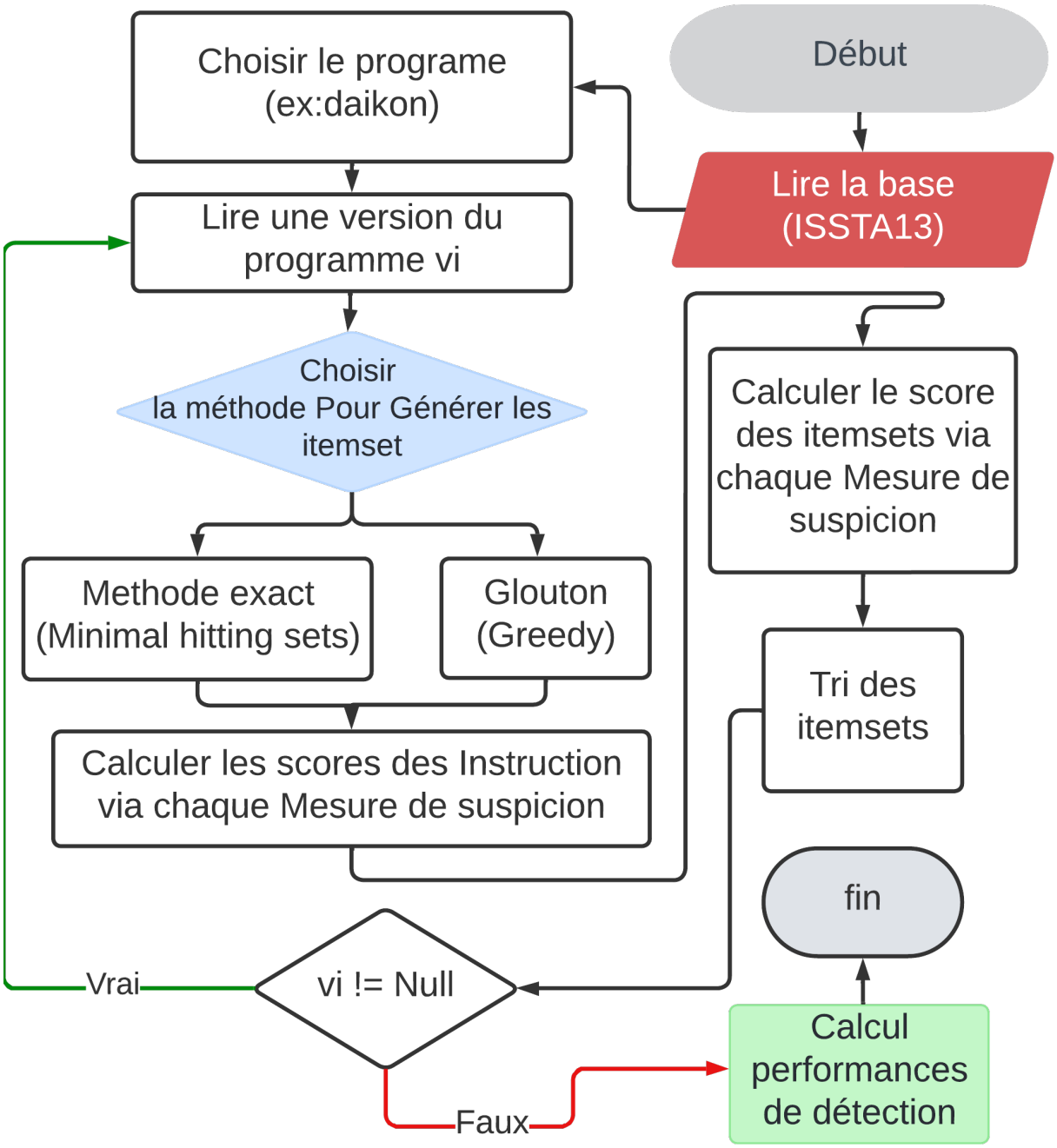

Figure 3.1: Organigramme de nos approches proposées.

3.3.2 Approche basée sur l'algorithme glouton

L'algorithme glouton (greedy algorithm en anglais) est un algorithme d'optimisation, qui construit une solution pas à pas en effectuant à chaque pas le meilleur choix possible. Dans le contexte de la localisation des fautes dans les programmes, l'algorithme construit des itemsets discriminants de manière itérative, en sélectionnant aléatoirement les éléments les plus pertinents.

Il existe trois types d'algorithmes glouton, glouton de haut en bas, glouton de bas en haut et glouton hybride, chacun avec un principe de fonctionnement distinct.

3.3.2.1 Glouton de bas en haut (bottom Up)

L'algorithme glouton bottom up (Algorithme 2), choisit un attribut aléatoirement (f) de la liste d'attributs (F_B) , le rajoute à l'ensemble S, et vérifie si cet attribut permet de discriminer les deux ensembles d'instructions négatives et positives($B+(S) \cap B-(S) = \emptyset$), si oui, l'algorithme s'arrête sinon, il continue de rajouter des attributs (choisis aussi aléatoirement) à S jusqu'à ce qu'il trouve un ensemble discriminant (Support set).

Données : Jeu de donnes binaire $B = (B^+, B^-)$
1 $S = \emptyset$;
2 **début**
3 | **tant que** *!supportSet(S)* **faire**
4 | | $f = SelectNextFeature(S, F_b)$;
5 | | $S = S \cup \{f\}$;
6 | **fin**
7 | **retourner** S;
8 **fin**

Algorithme 2 : L'algorithme glouton-bottom-up

3.3.2.2 Glouton de haut en bas (top Down)

L'algorithme top down (voir l'algorithme 3) commence par considérer tous les attributs de la base (F_B) dans S, puis tester si l'ensemble discrimine les deux classes d'instructions positives et négatives $B+(S) \cap B-(S) = \emptyset$), si c'est le cas , il enlève un attribut f de S aléatoirement, si l'ensemble est toujours discriminant, il continue la même procédure, sinon il remet cet élément dans S et choisit un autre aléatoirement. Le processus est répété jusqu'à ce qu'à épuisement de tous les attributs (tous les attributs déjà testés).

Données : Jeu de donnes binaire $B = (B^+, B^-)$
1 $S = F_B$;
2 **début**
3 | **tant que** *supportSet*(S) **faire**
4 | | $f = SelectNextFeature(S, F_B)$;
5 | | **si** *supportSet*$(S \backslash \{f\})$ **alors**
6 | | | $S = S \backslash \{f\}$;
7 | | **fin**
8 | **fin**
9 | **retourner** S;
10 **fin**

Algorithme 3 : L'algorithme glouton-up-bottom

3.3.2.3 Glouton hybride

L'algorithme hybride, comme son nom l'indique, est une hybridation des deux algorithmes cités précédemment, il s'agit de commencer d'abord par appliquer l'algorithme glouton de type bottom up puis l'algorithme glouton top down pour trouver ensemble d'attribut S. Ceci est décrit par algorithme 4 présenté ci-dessous.

Données : Jeu de donnes binaire $B = (B^+, B^-)$
1 **début**
2 | S = glouton-bottom-up(B,S);
3 | S = glouton-up-bottom(B,S);
4 | **retourner** S;
5 **fin**

Algorithme 4 : L'algorithme glouton-Hybride

Parmi ces trois méthodes, nous adoptons dans nos expérimentations la méthode hybride qui est une amélioration des deux autres.

3.3.3 Approche basée sur la méthode exacte

Notre deuxième approche est basée sur l'algorithme exactE (génération des traverses minimales ou minimal Hitting Set en anglais), qui permet de générer les ensembles minimaux de supports.

L'algorithme se compose principalement de deux étapes [F Delorme et al., 2024]

Etape 1 : transformation de la base de données de test en une base de données XOR. Cette dernière est ensuite transformée en un hypergraphe H = (N, A). où N est l'ensemble des instructions, et A l'ensemble des hyper arêtes. Chaque hyper arête représente une ligne de la base XOR restreinte aux instructions ayant une valeur 1 sur la ligne XOR.

Pour comprendre ce processus, nous prenons à titre d'exemple un petit programme [Maamar et al., 2017] , qui est illustré dans la table 3.2 présenté ci-dessous :

Table 3.2: Exemple des exécutions des instructions d'un petit programme [Maamar et al., 2017]

	cas de test							
inst	tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8
e1	1	1	1	1	1	1	1	1
e2	1	1	1	1	1	1	0	1
e3	1	1	1	1	1	1	0	0
e4	1	1	1	1	1	0	0	1
e5	1	1	0	0	1	0	0	0
e6	1	1	1	1	0	0	0	1
e7	0	1	0	1	0	0	0	0
e8	1	0	1	0	0	0	0	1
e9	1	0	1	0	0	0	0	1
e10	1	1	1	1	1	1	1	1
Positif/Négatif	N1	N2	N3	N4	N5	N6	P1	P2

Le produit cartésien entre la base B^+(positive Pi) et B^-(négative Ni) avec l'opération XOR est illustré dans la table 3.3 . Cette table comporte les instructions ei en colonnes et les valeurs calculées par l'équation en lignes :

$$(NiPi), ei = (Ni[ei]) \otimes (Pi[ei]) \tag{3.1}$$

Table 3.3: Base de données XOR générée

inst	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1N1	0	1	1	1	1	1	0	1	1	0
P1N2	0	1	1	1	1	1	1	0	0	0
P1N3	0	1	1	1	0	1	0	1	1	0
P1N4	0	1	1	1	0	1	1	0	0	0
P1N5	0	1	1	1	1	0	0	0	0	0
P1N6	0	0	1	0	0	0	0	0	0	0
P2N1	0	0	1	0	1	0	0	0	0	0
P2N2	0	0	1	0	1	0	1	1	1	0
P2N3	0	0	1	0	0	0	0	0	0	0
P2N4	0	0	1	0	0	0	1	1	1	0
P2N5	0	0	1	0	1	1	0	1	1	0
P2N6	0	0	1	1	0	1	0	1	1	0

Une fois que la base XOR est générée (voir table 3.3), l'hypergraphe est construit en considérant les cases comportant des 1 dans chaque ligne de la table, qui en réalité représente une hyper arrêt du graphe. L'hypergraphe généré correspondant à l'exemple présenté dans les tables 3.2 et 3.3 est le suivant :

e2 e3 e4 e5 e6 e8 e9

e2 e3 e4 e5 e6 e7

e2 e3 e4 e6 e8 e9

e2 e3 e4 e6 e7

e2 e3 e4 e5

e2 e3

e3 e5

e3 e5 e7 e8 e9

e3

e3 e7 e8 e9

e3 e5 e6 e8 e9

e3 e4 e6 e8 e9

Etape 2 : Calcul des Hitting sets minimaux de H [Murakami and Uno, 2013].

Étant donné une collection $H = \{H1, \ldots, Hm\}$ de sous-ensembles de V (ensemble d'instructions). H est aussi appelé un hypergraphe où chaque élément H_i de H est une hyper arête. Un hitting set (transversal) T de H est un sous ensemble de V qui intersecte chaque ensemble H_i de H ($T \cap Hi \neq \emptyset$) . T est dit Minimal Hitting Set (MHS) de H si aucun sous-ensemble de T n'est un hitting set de H.

Hitting sets : {e3}, {e3, e2} ; {e3, e4}, {e3,e5}, {e3, e6}, {e3, e7},{e3, e8}, {e3, e9}.....

Dans cet ensemble de hitting set, seul {e3} est considéré comme un minimal hitting set.

3.3.4 Ordonnancement des itemset générés

Une étape commune dans nos deux approches consiste à effectuer un ordonnancement des minimal hitting set dans l'objectif d'améliorer la localisation de la faute. Ceci est réalisé via le score calculé grâce aux mesures de suspicion (voir section 3.5) en appliquant la formule suivante :

$$score(itemset) = \frac{\sum Score(ei)}{Taille(itemset)^2} \tag{3.2}$$

Le score le plus élevé correspond au hitting set le plus susceptible de contenir la faute. Dans l'exemple présenté dans la section précédente, il s'agit d'un seul minimal hitting set {e3}.

3.3.5 Mesure de suspicion

La notion de suspicion dans le contexte des tests logiciels est étroitement liée à la nature des cas de test qui ont exécuté le programme. Lorsqu'une déviation du résultat attendu se produit, l'objectif est d'identifier le point précis du code où cette déviation a eu lieu. Cette déviation est le signe d'une exécution négative du test.

Dans ce contexte, il devient pertinent d'examiner attentivement les informations relatives à cette exécution négative. C'est pourquoi une partie du programme sous test est considérée comme suspecte, si cette partie du code est souvent exécutée par une exécution négative. Cela suggère que cette partie du code pourrait être la cause de la défaillance observée. Étant donné, un ensemble de cas de test positifs et négatifs, une instruction dans un programme est dite suspecte si elle est majoritairement exécutée par les cas de test négatifs et rarement par les cas de test positifs.

Parmi les méthodes exploitant cette notion de suspicion pour classer les instructions d'un programme, sont les techniques basées sur les mesures de suspicion où constituent un outil précieux pour le débogage logiciel. En attribuant un score de suspicion à chaque instruction d'un programme, ces techniques permettent de classer les instructions par ordre décroissant de leur implication potentielle dans les défaillances observées.

La plupart de ces mesures reposent sur l'hypothèse que les instructions les plus suspectes sont celles qui sont exécutées plus fréquemment par les cas de test négatifs que par les cas de test positifs.

Nous exposerons dans ce qui suit, les mesures de suspicion les plus populaires que nous avons adoptées, en plus de ça nous avons ajouté deux nouvelles mesures, le **XOR** et le **N et !P**. Chacune d'entre elles propose une méthode différente pour calculer le degré de suspicion d'une instruction en fonction de son exécution par les cas de test, en particulier les cas de test négatifs.

Les mesures de suspicion illustrées ci-dessous représentent une partie commune entre nos deux approches proposées (voir organigramme en section). Ces mesures sont utilisées pour classer les supports sets. La fonction XOR est exceptionnellement également utilisée pour générer l'hypergraphe exploité dans la méthode exacte.

3.3.5.1 Mesures de suspicion adoptées

1. **Tarantula**: c'est la mesure de suspicion la plus connue[Jones and Harrold, 2005], sa particularité est qu'elle se distingue par sa faible tolérance vis-à-vis des instructions fautives qui sont également exécutées par des cas de test positifs. Le degré de suspicion (score) affecté par Tarantula pour une instruction **ei** est calculé comme suit :

$$Tarantula(ei) = \frac{|\frac{negatif(ei)}{negatif(T)}|}{|\frac{positif(ei)}{positif(T)}| + |\frac{negatif(ei)}{negatif(T)}|} \tag{3.3}$$

Avec positif(T) (respectivement, négatif(T)) est l'ensemble de tous les cas de test positifs (respectivement. négatifs). positif(ei) (resp. négatif(ei)) est l'ensemble des cas de test positifs (resp. négatifs) couvrant ei .

2. **Ouchiai**: cette mesure est connue dans le domaine de la biologie [Abreu et al., 2007] . La particularité d'Ouchiai réside dans sa tolérance moyenne vis-à-vis des instructions fautives qui sont également couvertes par des cas de test positifs. Cela signifie qu'Ouchiai attribue un degré de suspicion modéré à de telles instructions, contrairement à des mesures comme Tarantula qui les pénalisent fortement. Le degré de suspicion (score) affecté par ouchia pour une instruction **ei** est la suivant :

$$Ouchiai(ei) = \frac{|negatif(ei)|}{|\sqrt{(|negatif(ei)| + |positif(ei)|) \times |negatif(T)|}} \tag{3.4}$$

3. **Jaccard** : la particularité de cette mesure est quelle est moins tolérante à la présence d'une instruction dans les cas de test positifs que la mesure Ochiai .Cette mesure a été présentée également dans [Abreu et al., 2007] avec l'hypothèse qu'une instruction exécutée principalement lors des cas de test positifs est moins suspecte qu'une instruction exécutée par les deux types de cas de test .Le degré de suspicion (score) affecté par Jaccard pour une instruction **ei** est la suivant :

$$Jaccard(ei) = \frac{|negatif(ei)|}{(|negatif(T)| + |positif(ei)|)} \tag{3.5}$$

4. **GP13** (Genetic Programming 13) : il existe plusieurs versions de GP, la plus utilisée et reconnu d'être efficace est la GP13 [Yoo, 2012], le degré de suspicion (score) affecté par cette mesure pour une instruction **ei** est la calculé comme suit :

$$GP13(ei) = Negatif(ei) \times \Big(1 + \frac{1}{\big((2 \times Positif(ei)) + Negatif(ei)\big)}\Big) \tag{3.6}$$

3.3.5.2 Mesures introduites

Afin d'optimiser nos résultats, c'est-à-dire obtenir le meilleur trie (ranking) possible des supports set (algorithme de glouton) ou des minimal hitting set (méthode exacte) nous avons également introduit deux autres nouvelles mesures, il s'agit du **XOR** et du **N et !P**.

1. **Mesure XOR**

   La première étape consiste à créer un table de produit cartésien entre les tests positifs P_i et les tests négatifs Ni du programme de base (voir Table 3.3).

   La deuxième étape consiste à déterminer le degré de suspicion (score), pour chaque

instruction ei en utilisant la mesure XOR. Ce score est calculé en additionnant le nombre de 1 dans la colonne correspondante à l'instruction **ei** (voir Table 3.4).

Table 3.4: Score calculé à partir de la table XOR générée (Table 3.3)

inst	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1N1	0	1	1	1	1	1	0	1	1	0
P1N2	0	1	1	1	1	1	1	0	0	0
P1N3	0	1	1	1	0	1	0	1	1	0
P1N4	0	1	1	1	0	1	1	0	0	0
P1N5	0	1	1	1	1	0	0	0	0	0
P1N6	0	0	1	0	0	0	0	0	0	0
P2N1	0	0	1	0	1	0	0	0	0	0
P2N2	0	0	1	0	1	0	1	1	1	0
P2N3	0	0	1	0	0	0	0	0	0	0
P2N4	0	0	1	0	0	0	1	1	1	0
P2N5	0	0	1	0	1	1	0	1	1	0
P2N6	0	0	1	1	0	1	0	1	1	0
score	0	5	12	6	6	6	4	6	6	0

2. **Mesure N et !P**

L'introduction de la mesure N et !P est motivé par le fait que cette dernière ne considère qu'une seul condition ($P_i = 0$ et $N_i = 1$) pour identifier une instance positive, tandis que la méthode XOR exige la présence des deux conditions ($P_i = 0$ et $N_i = 1$) et ($P_i = 1$ et $N_i = 0$), ce qui la rend plus sensible au bruit.

Nous donnons dans la table 3.5 le produit cartésien entre les tests positifs P_i et les tests négatifs N_i du programme de base avec l'opération N et !P.

Cette table comporte les instructions **ei** en colonnes, et dans les lignes les valeurs calculées par l'équation suivante :

$$(NiPi), ei = (Ni(ei)) \cap \overline{(Pi(ei))} \tag{3.7}$$

Le degré de suspicion (score) pour chaque instruction ei en utilisant la mesure N et !P représenté également dans la table 3.5, est calculé en additionnant le nombre de 1 dans la colonne correspondante à l'instruction **ei**.

Table 3.5: Score calculé via la mesure N et !P sur l'exemple de la table 3.2

inst	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1N1	0	1	1	1	1	1	0	1	1	0
P1N2	0	1	1	1	1	1	1	0	0	0
P1N3	0	1	1	1	0	1	0	1	1	0
P1N4	0	1	1	1	0	1	1	0	0	0
P1N5	0	1	1	1	1	0	0	0	0	0
P1N6	0	1	1	0	0	0	0	0	0	0
P2N1	0	0	1	0	1	0	0	0	0	0
P2N2	0	0	1	0	1	0	1	0	0	0
P2N3	0	0	1	0	0	0	0	0	0	0
P2N4	0	0	1	0	0	0	1	0	0	0
P2N5	0	0	1	0	1	0	0	0	0	0
P2N6	0	0	1	0	0	0	0	0	0	0
score	0	6	12	5	6	4	4	2	2	0

Nous donnons dans la Table 3.6 les valeurs (susp) des degrés de suspicion calculée via les différentes mesures de suspicion illustrées précédemment, sur chaque instruction du programme. Nous présenterons également dans cette table, le classement (Rang) retourné, pour chaque type de mesures.

Table 3.6: Degrés de suspicion selon différentes mesures sur le même programme (table 3.2).

	mesure de suspecion											
inst	XOR		N et !P		TARANT		JACC		OCHI		GP13	
	susp	Rang	susp	Rang	susp	Rang	susp	Rang	susp	Rang	susp	Rang
e1	0	10	0	10	0.5	8	0.75	4	0.86	4	6.6	4
e2	5	7	6	3	0.66	4	0.85	2	0.92	2	6.75	2
e3	12	1	12	1	1	1	1	1	1	1	7	1
e4	6	6	5	4	0.62	5	0.71	5	0.83	5	5.71	5
e5	6	6	6	3	1	1	0.5	7	0.70	7	4	7
e6	6	6	4	6	0.57	6	0.57	6	0.73	6	4,64	6
e7	4	8	4	6	1	1	0.33	8	0.57	8	3	8
e8	6	6	2	8	0.4	10	0.28	10	0.47	10	2.5	10
e9	6	6	2	8	0.4	10	0.28	10	0.47	10	2.5	10
e10	0	10	0	10	0.5	8	0.75	4	0.86	4	6.6	4

3.4 Résultats et discussions

Durant la phase test et validation de nos approches proposées, nous avons utilisé la base de données nommée ISSTA13 ("International Symposium on Software Testing and Anal-

ysis",Year 2013) lien de la base , qui est composée de dix programmes, ou chaque programme a plusieurs versions, avec des emplacements de fautes variés.

Afin de mesurer l'efficacité de nos approches, nous avons utilisé des mesures d'évaluation des performances, à savoir l'exam score [Wong et al., 2011] dans ses versions pessimiste, optimiste et delta exam score

3.4.1 Mesure d'évaluation des performances

3.4.1.1 Exam score pessimiste

C'est une mesure qui reflète le pourcentage d'instructions non fautives vérifiées avant de trouver la faute, en considérant la pire performance. Elle est calculée via la formule suivante :

$$PExam(ProgrammeBase) = \frac{Pascand(ei)}{Taille(T)} \times 100 \tag{3.8}$$

Avec :

- Pascand(ei) est une fonction que nous avons implémentée qui compte le nombre d'instructions dont la position est avant l'instruction contenant la faute(on supposant que l'erreur est toujours en dernière position dans l'itemsets qu'elle appartient)
- Taille (T) représente le nombre total des instructions du programme.

3.4.1.2 Exam score optimiste

Cette une mesure qui reflète le pourcentage d'instructions non fautives vérifiées avant de trouver la faute, en considérant la meilleure performance. Elle est calculée via la formule suivante :

$$OExam(ProgrammeBase) = \frac{Oascand(ei)}{Taille(T)} \times 100 \tag{3.9}$$

Avec :

- Oascand est une fonction que nous avons implémentée qui compte le nombre d'instruction dont la position (après tri), se trouve avant l'itemset contenant l'erreur.

3.4.1.3 Delta exam score

Le Delta score représente une valeur moyenne entre le l'exam score optimiste et l'exam score pessimiste.

$$DExam(ProgrammeBase) = \frac{Dascand(ei)}{Taille(T)} \times 100 \tag{3.10}$$

Avec:

- Dascand(ei) est une fonction que nous avons implémentée, qui compte le nombre d'instructions situés avant l'instruction contenant l'erreur après l'étape d'ordonnancement des itemset , où on suppose que l'erreur au milieu de l'itemset dont elle appartient.

Avant de discuter les résultats, nous rappelons que nous avons implémenté deux approches, la première basée sur l'algorithme glouton hybride et la seconde via l'algorithme exact. Nos approches de localisation de fautes s'appuient sur l'analyse logique des données.

3.4.2 Résultats de la première approche : algorithme Glouton Hybride

L'algorithme de Glouton permet de générer plusieurs ensembles d'items. Après plusieurs tests, nous avons fixé le nombre d'itérations de l'algorithme à 100, et ce, après élimination des redondances. Les mesures de suspicion jouent un rôle primordial dans nos approches, car elles permettent de classer les ensembles d'items générés. Nous espérons que l'erreur se trouve en tête du classement.

3.4.2.1 Calcul des scores des instructions

Il s'agit dans un premier temps de calculer les scores de chaque instruction, via six mesures de suspicion à savoir Tarantula , GP13,Jaccard , Ochiai et les deux mesures que nous avons introduit le XOR et la mesure N et !P . Ceci est illustré dans la table 3.7.

Table 3.7: Extrait des scores des instructions du programme Htmlparser de la base ISSTA13 lien de la base calculés via six mesures de suspicion

inst	XOR	N et !P	TARANTULA	JACCARD	OCHIAI	GP13
1	404	404	0.75	0.07	0.01	1
2	200	200	0.6	0.05	0	1
3	197	197	0.6	0.05	0	1
4	7	0	0	0	0	0
5	194	194	0.6	0.05	0	1
6	1	0	0	0	0	0
7	41	0	0	0	0	0
8	311	311	0.68	0.06	0	1
9	4	0	0	0	0	0
10	596	596	1	0.5	0.25	1.14
11	1	0	0	0	0	0
12	194	194	0.6	0.05	0	1
..	..	..	..	..	..	..
..	..	..	..	..	..	..
363	20	0	0	0	0	0
364	44	0	0	0	0	0
$\Rightarrow$ 365$\Leftarrow$	598	598	1	0.71	0.5	1.33
366	194	194	0.6	0.05	0	1
..	..	..	..	..	..	..
..	..	..	..	..	..	..
782	6	0	0	0	0	0
783	1	0	0	0	0	0
784	194	194	0.6	0.05	0	1

3.4.2.2 Calcul des scores des item set générés via Glouton hybride

Dans un deuxième temps, pour chaque Itemset généré, nous calculons le score selon les six différentes mesures en utilisant les résultats illustrés dans la table 3.7 contenant les scores des instructions. Nous avons illustré dans les tables 3.8 et 3.9 des extraits de classement des itemsets issus du programme Htmlparser de la base ISSTA13 lien de la base respectivement via les scores XOR, et GP13.

Table 3.8: Extrait de classement des itemsets générés via Glouton hybride à partir du programme Htmlparser de la base ISSTA13 lien de la base via leur score XOR .

itemset	XOR	N et !P	TARANTULA	JACCARD	OCHIAI	GP13
255	598	598	1	0,71	0,5	1,33
380 $\Rightarrow$ 365$\Leftarrow$	177	149,5	0,25	0,1775	0,125	0,3325
255 380	177	149,5	0,25	0,1775	0,125	0,3325
380 281	177	149,5	0,25	0,1775	0,125	0,3325
565 $\Rightarrow$ 365$\Leftarrow$	166	149,5	0,25	0,1775	0,125	0,3325
281 565	166	149,5	0,25	0,1775	0,125	0,3325
584 255	161,5	149,5	0,25	0,1775	0,125	0,3325
255 592	157,25	149,5	0,25	0,1775	0,125	0,3325
281 592	157,25	149,5	0,25	0,1775	0,125	0,3325
460 281	157	149,5	0,25	0,1775	0,125	0,3325
460 $\Rightarrow$ 365$\Leftarrow$	157	149,5	0,25	0,1775	0,125	0,3325
$\Rightarrow$ 365$\Leftarrow$ 314	157	149,5	0,25	0,1775	0,125	0,3325
255 460	157	149,5	0,25	0,1775	0,125	0,3325
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
235	8	0	0	0	0	0
777	1	0	0	0	0	0

On remarque que le trie selon les scores calculés par la mesure XOR illustrés par la table 3.8, permet de classer l'Itemset contenant l'erreur dans la 2^{eme} position. Sachant que l'instruction contenant l'erreur dans ce programme selon l'étiquette de la base est “ 365”.

Table 3.9: Extrait de classement des itemsets issus du programme Htmlparser de la base ISSTA13 lien de la base via la mesure de suspicion GP13.

itemset	XOR	N et !P	TARANTULA	JACCARD	OCHIAI	GP13
255	598	598	1	0,71	0,5	1,33
281 565	166	149,5	0,25	0,1775	0,125	0,3325
281 235	151,5	149,5	0,25	0,1775	0,125	0,3325
⇒365⇐ 487	150,25	149,5	0,25	0,1775	0,125	0,3325
565 ⇒365⇐	166	149,5	0,25	0,1775	0,125	0,3325
255 592	157,25	149,5	0,25	0,1775	0,125	0,3325
281 592	157,25	149,5	0,25	0,1775	0,125	0,3325
460 ⇒365⇐	157	149,5	0,25	0,1775	0,125	0,3325
314 ⇒365⇐	157	149,5	0,25	0,1775	0,125	0,3325
235 255	151,5	149,5	0,25	0,1775	0,125	0,3325
⇒365⇐ 169	149,75	149,5	0,25	0,1775	0,125	0,3325
140 255	151,5	149,5	0,25	0,1775	0,125	0,3325
⋮	⋮	⋮	⋮	⋮	⋮	⋮
314	30	0	0	0	0	0
777	1	0	0	0	0	0

Selon ces résultats de classement des itemset (Table 3.8 et 3.9) via les mesures de suspicion (score), dans une version de programme Htmlparser, et en comparant les deux positions de l'erreur (2^{eme} pour le XOR et 4^{eme} position pour le GP13), nous remarquons que la mesure XOR est meilleure que la mesure GP13, car l'erreur est trouvée plus rapidement.

Le classement de l'erreur varie hélas d'une version d'un programme à une autre, et d'une mesure à une autre. Ceci est aussi visible suite au calcul de l'exam score pessimiste, optimiste et le delta, via les différentes mesures de suspicion (voir les Table 3.10 et 3.11, section 4.2.3).

3.4.2.3 Calcul de l'exam score

Afin de prouver l'efficacité de l'algorithme de Glouton dans sa variante hybride, nous avons également calculé les exams scores des différentes versions des dix programmes de la base ISSTA13 lien de la base , ceci est illustré sur la table 3.11.

Mais nous allons d'abord présenter dans la table 3.10 les résultats de quelques versions du programme evenbus de la base ISSTA13 lien de la base

Table 3.10: L'Exam scores pessimiste, optimiste et delta, de quelques versions du programme evenbus de la base ISSTA13 lien de la base via la méthode Glouton hybride

	Version	\v171	\v179	\v181	\v184	\v190	\v191
XOR	Eso	0,12837	0,254453	0,508906	0,256082	12,85141	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	13,25301	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	26,63989	4,551365
N et !P	Eso	0,12837	0,254453	0,508906	0,256082	10,84337	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	11,51272	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	21,15127	4,551365
Tarantula	Eso	0,12837	0,254453	0,508906	0,256082	10,97724	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	11,64659	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	21,95448	4,551365
Ochiai	Eso	0,12837	0,254453	0,508906	0,256082	10,97724	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	11,64659	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	21,95448	4,551365
Jaccard	Eso	0,12837	0,254453	0,508906	0,256082	11,51272	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	12,18206	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	23,1593	4,551365
Gp13	Eso	0,12837	0,254453	0,508906	0,256082	10,97724	1,040312
	Esp	0,12837	0,381679	0,636132	0,384123	11,64659	1,040312
	EsD	0,12837	0,381679	0,636132	0,384123	21,95448	4,551365

Selon les résultats illustrés dans la table 3.10, nous remarquons que les meilleures performances sont celles du XOR et de N et !P. Effectivement, les valeurs des trois versions de l'exam score obtenues pour ces mesures-là, sont les plus réduites par rapport aux autres mesures. Toutefois, rappelons que le l'exam score est une mesure qui reflète le pourcentage d'instructions à vérifier ne contenant pas d'erreurs avant de tomber sur l'instruction erronée, ce qui représente un effort fourni et un temps perdus.

Table 3.11: Exam score optimiste de quelle que programme de la base ISSTA13

program	XOR	N et !P	Tarantula	Ochiai	Jaccard	Gp13
daikon	0,7892	0,7892	0,8509	0,7684	0,7684	0,7684
eventbus	26,08246	25,6097	26,486	25,7855	25,5142	26,1076
jester1	14,0008	13,4021	13,8550	13,4951	13,1962	13,4634
codec	11,9108	11,8494	12,7383	11,5964	11,1959	11,8988
drow2d	6,2870	5,8102	5,9695	5,5468	5,4980	5,7657
htmlparser	6,8652	5,8650	6,0792	5,9100	5,9194	6,4437

3.4.3 Deuxième approche: algorithme exacte (Minimal Hitting Set (MHS))

L'algorithme Minimal Hitting Set, également connu sous le nom de "méthodes Exacte", est une approche exhaustive permettant la génération d'itemset minimal. Dans notre contexte, elle permet la découverte d'itemsets discriminants dans le cadre de l'analyse de données logiques. Il s'agit d'une méthode systématique qui explore toutes les combinaisons possibles d'instructions discriminantes, raison pour laquelle elle est nommée exacte.

L'objectif principal de l'utilisation de cette méthode est alors d'identifier l'ensemble complet des itemsets discriminants, sans aucune redondance. Cela signifie que chaque itemset découvert est unique et contribue de manière significative à la discrimination entre les exécutions réussies et échouées du programme, et par conséquent une grande possibilité pour détecter l'erreur. Cette possibilité, qui est justement augmentée grâce aux mesures de suspicion.

3.4.3.1 Limites et inconvénients de la méthode exacte

Malgré sa précision et son exhaustivité, la méthode exacte présente certaines limitations importantes :

- La complexité de l'algorithme est exponentielle par rapport à la taille maximale des itemsets. Cela signifie que le temps d'exécution et les ressources matérielles nécessaires augmentent de manière exponentielle à mesure que la taille des itemsets augmente, ce qui entraîne un coût computationnel.
- Pour les bases de données de taille moyenne, la méthode exacte peut générer un nombre colossal d'itemsets, ce qui rend l'analyse et l'interprétation des résultats difficiles et chronophages.
- Problèmes de mémoire : la génération et le stockage de tous les itemsets discriminants peuvent rapidement dépasser les capacités de la mémoire disponible, en particulier pour les bases de données volumineuses.
- Infaisabilité pratique pour les grandes bases de données: Pour les bases de données de grande taille, l'exécution de la méthode exacte peut devenir impraticable en raison des contraintes de temps et de ressources.

3.4.3.2 Exemples d'itemsets générés

En raison des limitations de la méthode exacte (voir section 4.3.1), il a été nécessaire de limiter la taille maximale des itemsets à 5 instructions. Cette restriction a permis de générer un nombre gérable d'itemsets et d'obtenir un aperçu des résultats de la méthode.

A titre d'exemple, nous présentons un extrait des résultats obtenus à partir du programme daikon via la méthode exacte. Dans cet exemple, la méthode a généré exactement 8390942 itemsets (voir la figure 3.2).

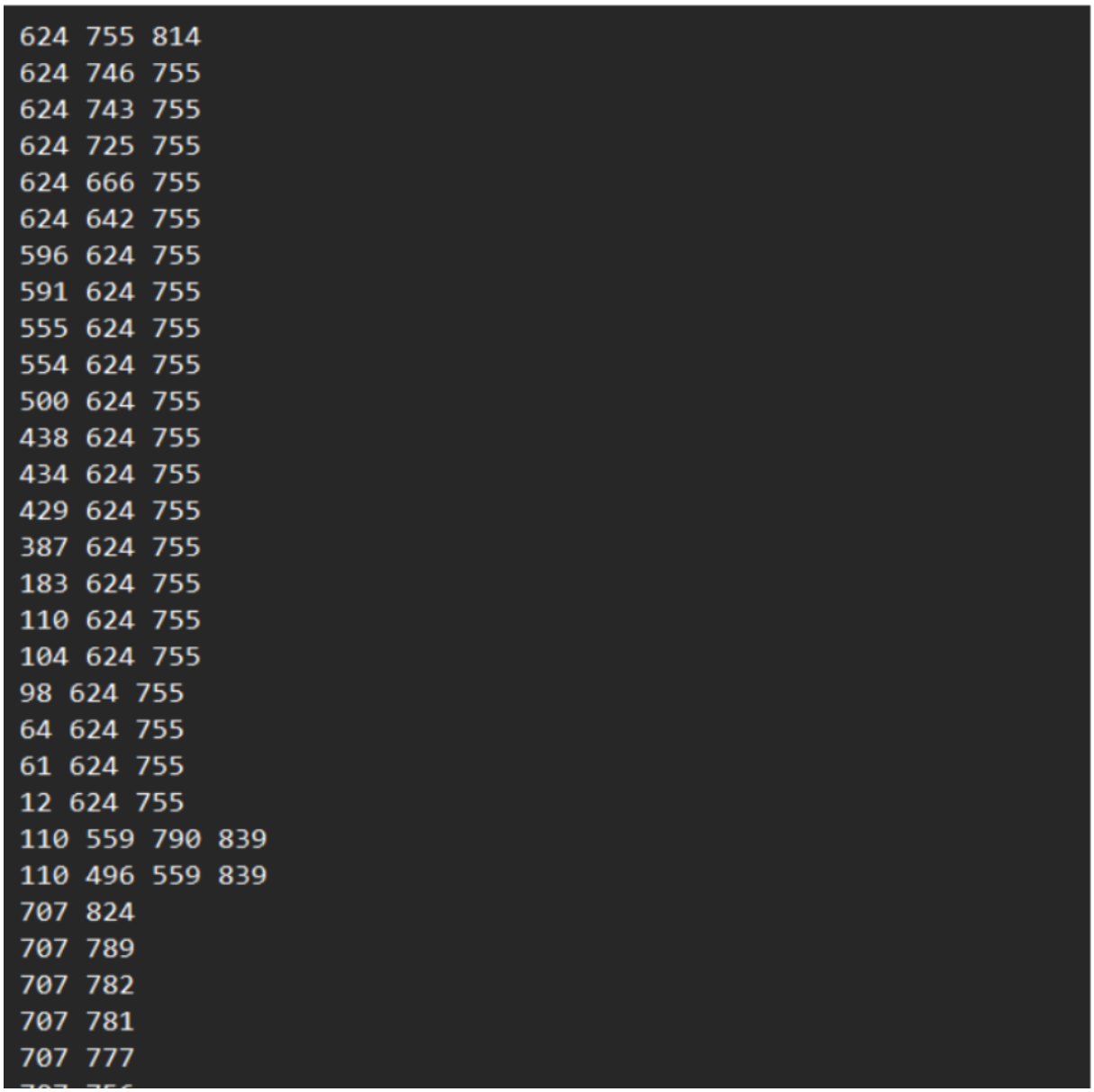

Figure 3.2: Extrait des itemsets générés de taille <=5 à partir du programme daikon via la méthode exacte

Il est important de noter que la figure 3.2 ne représente qu'un échantillon des itemsets discriminants potentiels. Si la limitation de la taille des itemsets n'avait pas été appliquée, l'algorithme aurait pu générer un nombre beaucoup plus important d'itemsets et de taille plus grande.

3.4.3.3 Implications pratiques de la complexité exponentielle

Même si la méthode exacte permet de générer tous les itemsets discriminants sans redondance, il est important de souligner que le calcul des scores de chaque itemset avec chaque mesure de suspicion peut devenir impraticable pour les bases de données de grande taille.

En effet, la complexité exponentielle de l'algorithme implique que le temps de calcul nécessaire pour évaluer tous les itemsets possibles augmente de manière exponentielle avec la taille de la base de données. Cela signifie que même si les ressources matérielles nécessaires sont disponibles, le calcul des scores peut prendre un temps prohibitif, rendant la méthode inapplicable dans des contextes réalistes.

3.5 Implémentation et présentation de l'application

3.5.1 Matériel utilisé

Dans le cadre de cette étude, deux types d'ordinateurs ont été utilisés : un ordinateur portable où nous avons testé et exécuté notre approche basée sur l'algorithme glouton Hybride, et un ordinateur de bureau dédié à l'exécution de la méthode exacte, nécessitant des ressources plus performantes.

Les caractéristiques de l'ordinateur portable :

- CPU :Processeur 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz , 4 cœur(s), 8 processeur(s) logique(s)
- Mémoire physique (RAM) : 16,0 Go
- Système d'exploitation Windows 11 64 bits, processeur x64
- GPU: Intel Iris Xe Graphics

Les caractéristiques de l'ordinateur de bureau :

- CPU : Processeur 06th Gen Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz,4 cœur(s), 8 processeur(s) logique(s)
- Mémoire physique (RAM) : 08,0 Go
- Système d'exploitation Linux 64 bits, processeur x64
- GPU: dual carte graphique (Intel HD Graphics 530) et (NVIDIA GF 108[GeForce GT 730])

3.5.2 Environnement de développement

Ces environnement de développement met à profit les forces de Python, Django REST Framework et C# WPF pour offrir une solution complète pour les projets d'analyse de données complexes. **Python comme moteur d'analyse:**

- Python, avec ses bibliothèques spécialisées comme [NumPy, excelle dans le traitement, le nettoyage et l'analyse de données.
- Son langage simple et sa communauté active en font un choix accessible et bien supporté pour les data scientists.

Django REST Framework pour des API puissantes:

- Django REST Framework permet de créer des API RESTful claires, cohérentes et sécurisées pour exposer les résultats des analyses et interagir avec d'autres applications.
- Sa flexibilité permet de modeler les API selon les besoins spécifiques du projet.

C# WPF pour des interfaces utilisateur riches:

- C# WPF(en anglais Windows Presentation Foundation) permet de créer des interfaces utilisateur graphiques intuitives et performantes avec des animations fluides et des visualisations attrayantes.
- Son intégration native avec Windows facilite le développement et le déploiement sur des systèmes Windows.

Architecture clien de la baset-serveur pour une évolutivité accrue:

- La séparation du backend (Django REST Framework) et du frontend (C# WPF) favorise la scalabilité et la maintenance.
- Le déploiement indépendant des composants offre une flexibilité supplémentaire.

3.5.3 Présentation de notre application

Nous avons réalisé une interface pour notre programme afin de rendre son utilisation plus simple et ergonomique, il s'agit de l'IHM de l'approche Glouton. Dans ce qui suit une petite présentation de notre Application.

3.5.3.1 Fenêtre principale (accueil)

Cette fenêtre contient trois paramètres de traitement, et un bouton d'analyse (voir figure 3.3).

1. S**élection du programme:** un menu déroulant permet de choisir un programme parmi dix de base ISSTA13,

2. **Choix de l'algorithme:** deux algorithmes sont disponibles : glouton et la méthode exacte.

3. **Sélection de la source de données:** l'utilisateur peut choisir entre deux sources : locale (fichiers enregistrés sur l'ordinateur) et serveur (base de données distante pour des analyses en temps réel).

4. **Bouton "Analyse":** lance le processus d'analyse en utilisant les paramètres sélectionnés. L'analyse peut prendre quelques minutes, en fonction de la taille des données et des paramètres choisis.

Description: à gauche de la fenêtre présentée par la figure 3.3, nous avons un panneau contenant les menus : instructions, les itemsets et les résultats obtenus. Ce panneau est désactivé par défaut et devient actif uniquement après le lancement d'une analyse.

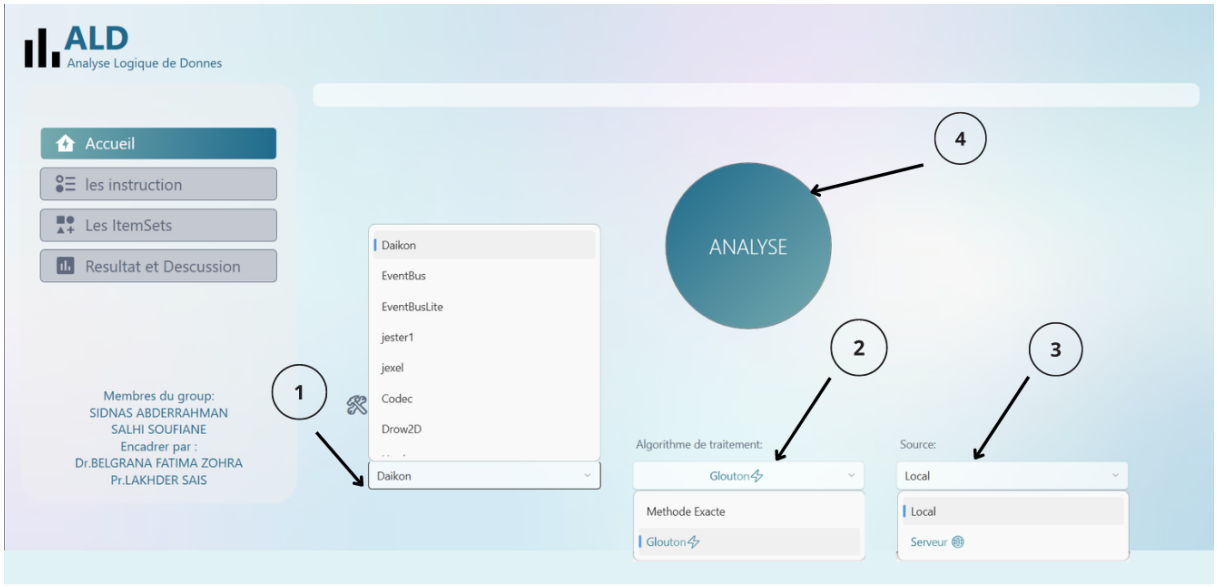

Figure 3.3: Fenêtre principale de notre application

- Une fois les paramètres de traitement définis, l'utilisateur peut lancer la détection en cliquant sur le bouton **"ANALYSE"**. Cette opération prend un certain temps, ceci est illustré par la figure 3.4.

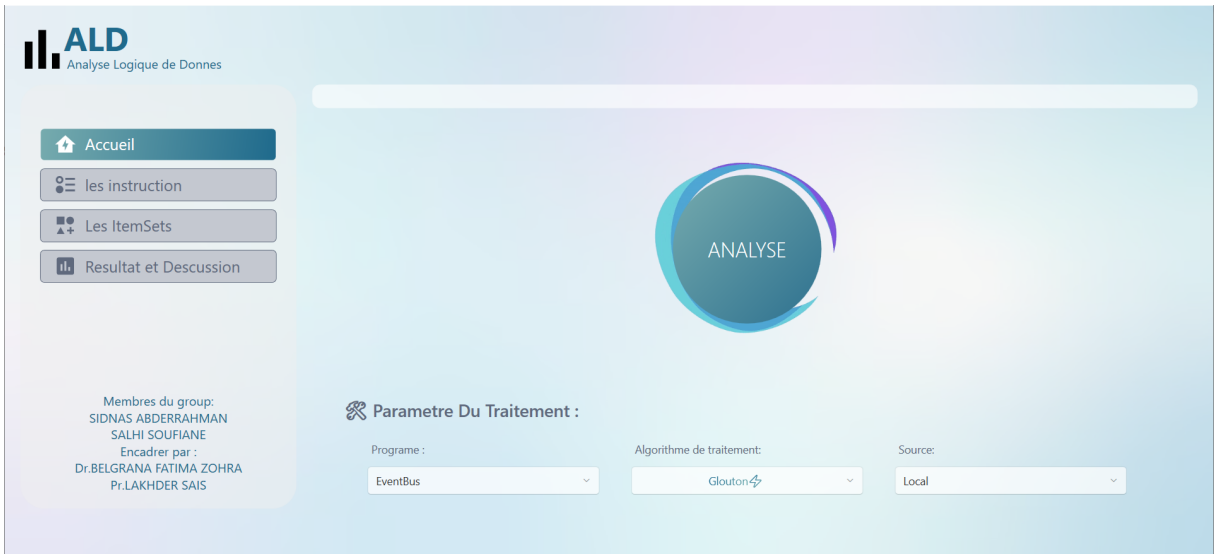

Figure 3.4: Définition des paramètres et lancement de l'analyse

- Une fois l'analyse terminée, le panneau de gauche sera activé et une notification de succès apparaîtra dans une boîte d'information (show message) contenant le message « Cliquez sur "OK" » (voir la figure 3.5)

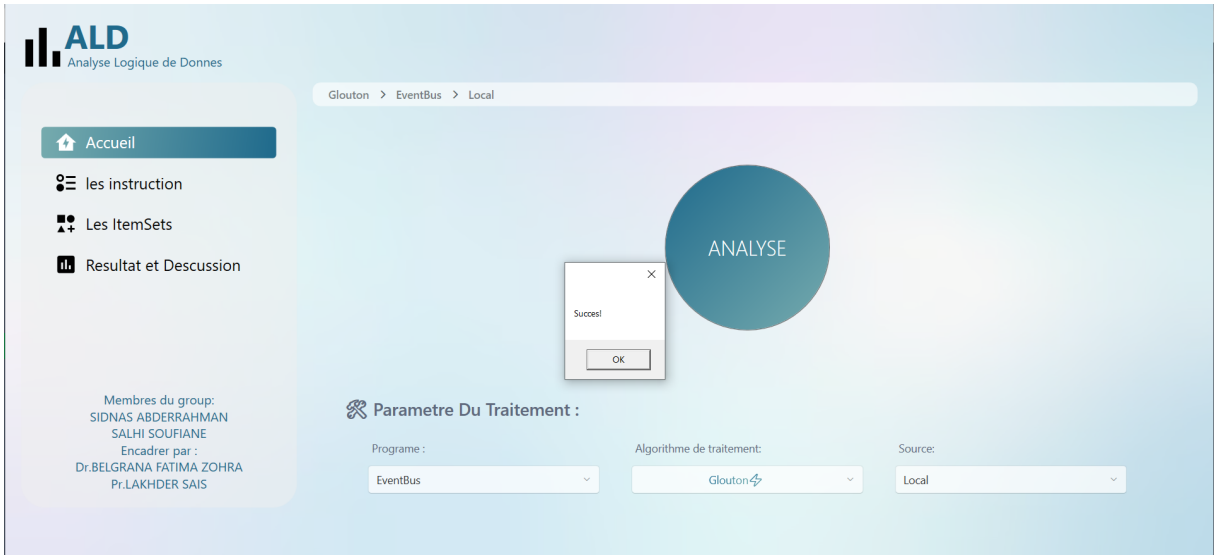

Figure 3.5: Fin de l'analyse

3.5.3.2 Fenêtre des instructions

La fenêtre d'instructions affiche une table complète des résultats de score de chaque instruction via les six mesures de suspicion. Chaque ligne de la table correspond à une instruction, tandis que les colonnes affichent différentes mesures de suspicion.

Deux paramètres supplémentaires sont disponibles :

1. **Choix de la version du programme :** permet d'altérer entre les versions d'un même programme

2. **Choix du critère de tri:** permet de définir le critère de tri des instructions dans la table. Par défaut, le tri est effectué par score, mais d'autres options sont disponibles pour une analyse plus précise.

La table affiche initialement les résultats de la première version du programme, triés par score (voir la figure 3.6). En bas de la table, un panneau d'informations fournit des détails supplémentaires sur chaque version du programme utilisée.

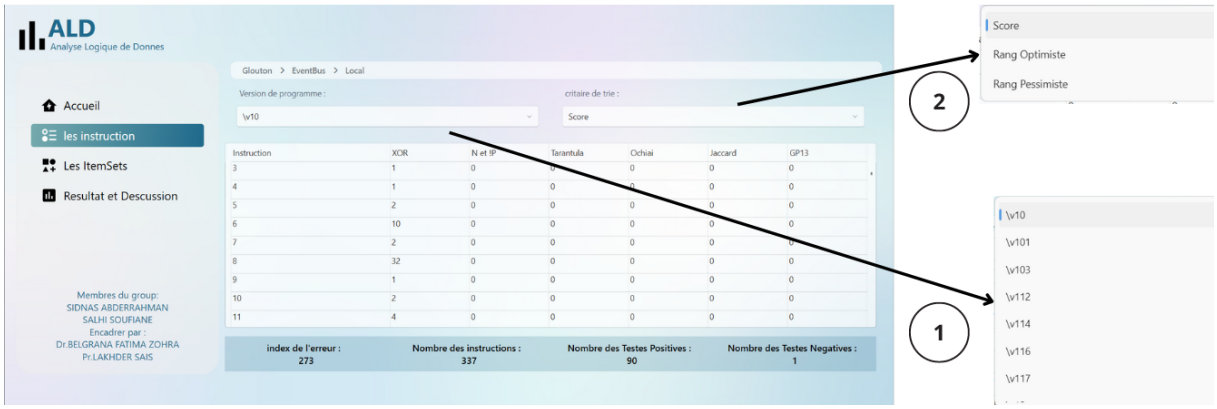

Figure 3.6: Affichage de la table des instructions non triées

- Le clique sur une colonne permet de désigner la mesure de suspicion avec laquelle on souhaite trier la liste des instructions. À titre d'exemple, le choix de la mesure XOR est illustré par la figure 3.7.

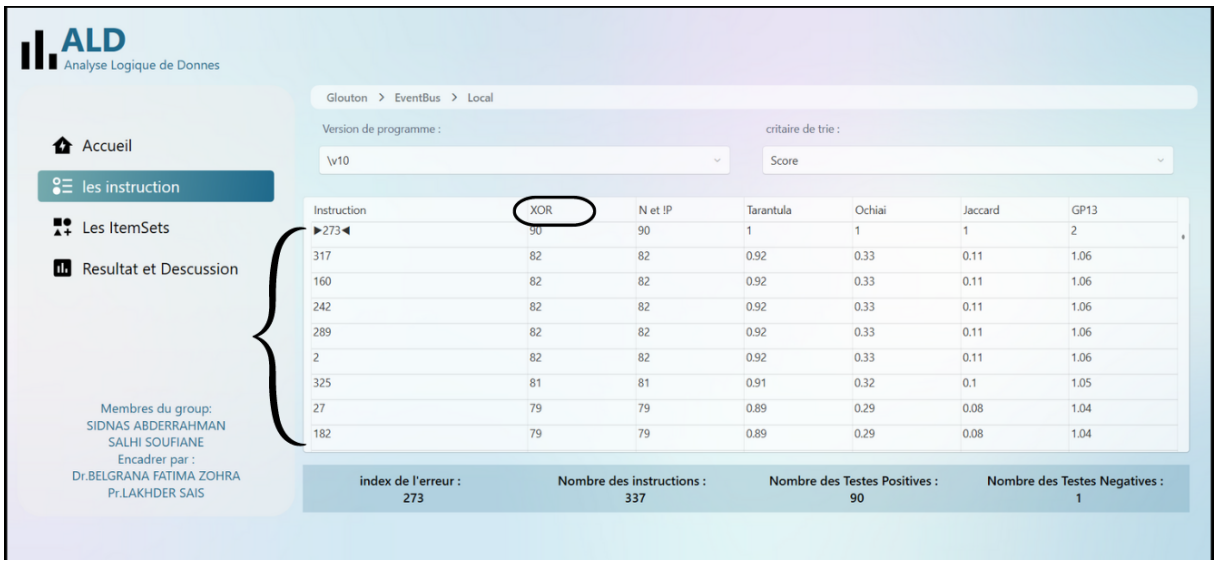

Figure 3.7: Affichage des instructions triées via la mesure XOR

3.5.3.3 Fenêtre des itemset

La fenêtre des itemset permet l'affichage d'une table contenant les résultats de score de chaque itemset générés selon les six différentes mesures, chaque ligne de la table correspond à un itemset, tandis que les colonnes affichent différentes mesures de suspicion (voir la

figure 3.8).

Cette fenêtre permet le choix d'un seul paramètre qui est la version du programme.

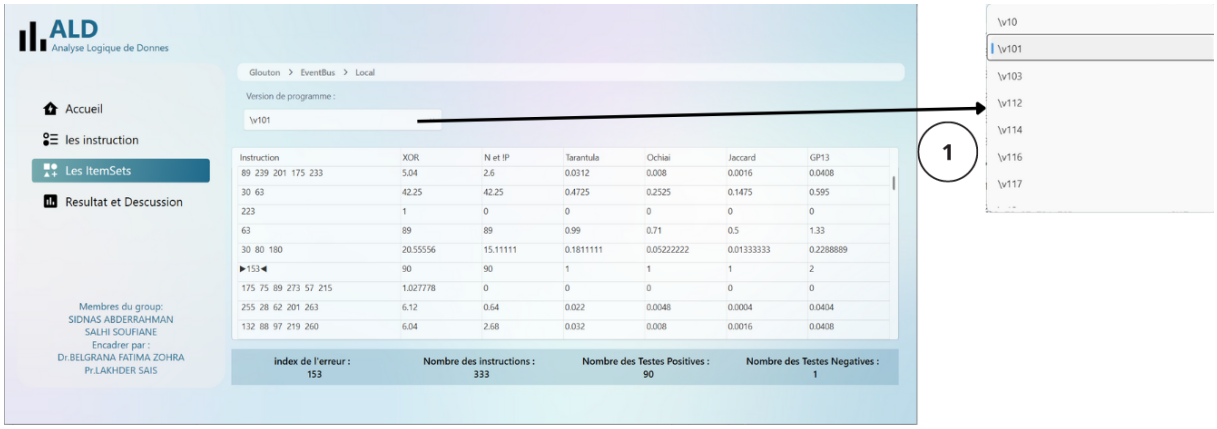

Figure 3.8: Affichage des itemset non triés

- La figure 3.9 présentée ci-dessous illustre le choix de la mesure Tarantula pour trier les itemset.

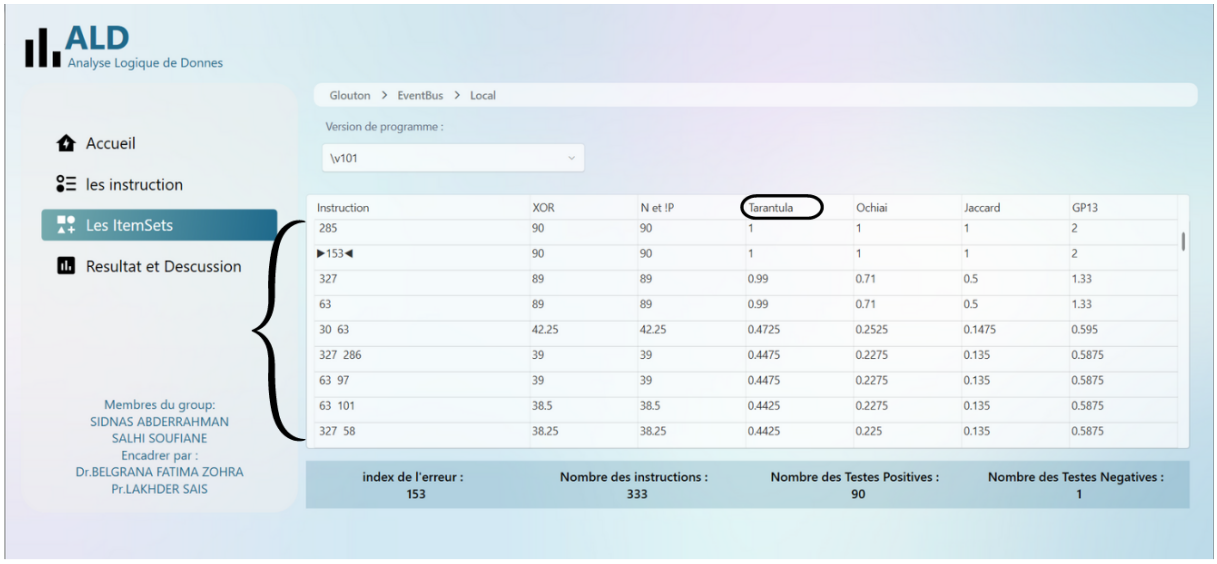

Figure 3.9: Affichage des itemset triés selon la mesure Tarantula

3.5.3.4 Fenêtre des résultats

Cette fenêtre permet l'affichage des résultats de l'exam score en choisissant une parmi ses trois versions : pessimiste, optimistes, ou delta exam score (voir Figure 3.10).

Cette fenêtre permet le choix entre trois formes d'affichage, une table, et deux diagrammes.

La table est illustrée dans la figure suivante. Un exam score petit indique une meilleure détection de l'instruction fautive.

Dans cette figure on remarque par exemple que l'exam score de la version V252 est plus petit que l'exam score de de la version V199 selon la mesure XOR. Ceci indique que notre approche Greedy permet de détecter la faute plus rapidement dans la version V252 que dans la version V199.

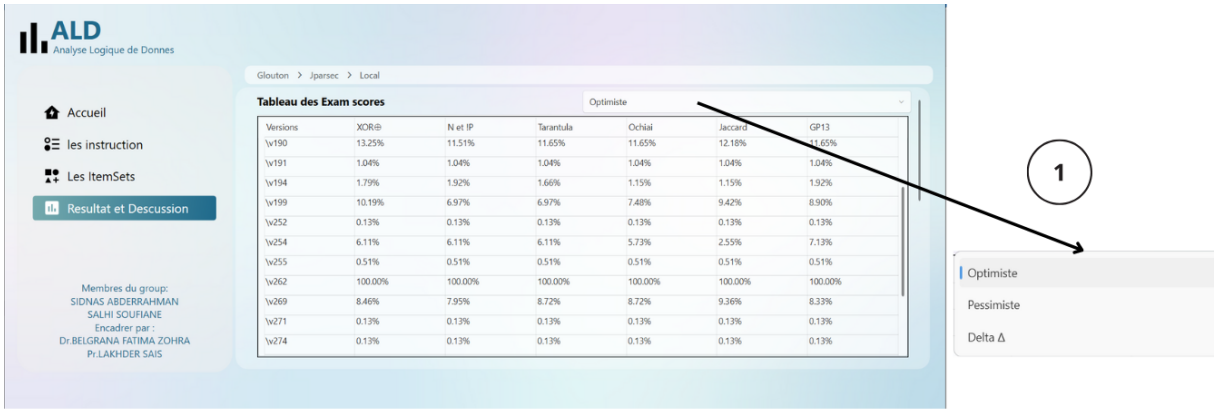

Figure 3.10: Affichage de l'exam score optimiste de chaque version

- Un nuage de points est généré pour comparer l'exam score des différentes versions pour chaque mesure de suspicion.ceci est illustré par la figure 3.11.

Figure 3.11: Affichage du nuage de points

- Un diagramme à barres est créé afin de comparer les moyennes d'exam score selon les différentes mesures de suspicion. (voir la figure 3.12)

Chaque barre du diagramme représente la moyenne d'exam score obtenus pour une mesure de suspicion spécifique.

La moyenne d'exam score est calculée en divisant la somme des exams scores de toutes les versions de programme par le nombre total de versions.

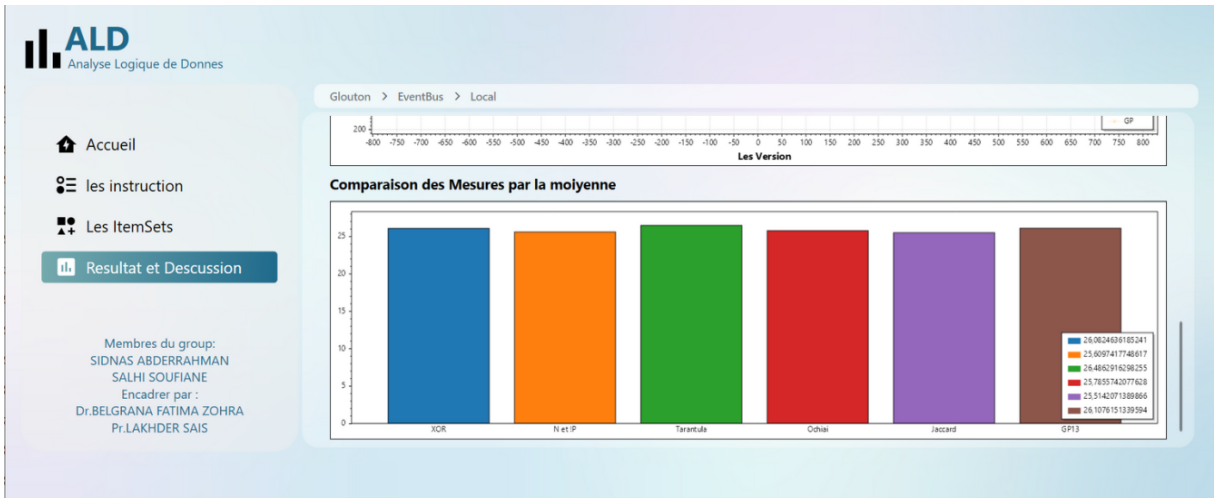

Figure 3.12: Affichage du diagramme à barres

3.6 Conclusion

Dans ce chapitre, nous avons présenté nos deux approches proposées de localisation de fautes dans les programmes, la première est basée sur l'algorithme de Glouton et la seconde sur la méthode exacte. Nos approches de localisation de fautes s'appuient sur l'analyse logique des données pour générer des ensembles de support où nous avons commencé par décrire le principe général de nos approches en fournissant un organigramme.

De plus, nous avons souligné l'importance des mesures de suspicion à savoir Tarantula , GP13,Jaccard et Ochiai, où nous avons introduit deux nouvelles mesures de suspicion, le XOR et le N et !P . Ces mesures sont utilisées afin de sélectionner le meilleur support parmi ceux qui sont générés et qui devrait contenir l'erreur.

Nous avions comme objectif de trouver une approche efficace et moins coûteuse en temps et en ressource pour détecter les fautes dans les programmes.

Afin de tester nos approches proposées, nous avons utilisé une base de données contenant plusieurs programmes avec diverses versions, contenant une seule faute à la fois.

Afin de prouver l'efficacité de nos approches, nous avons utilisé des mesures d'évaluation des performances, il s'agit de l'exam score avec ses trois versions pessimistes et optimistes, ainsi que le delta Exam score. Les résultats illustrés sont satisfaisants.

Conclusion générale

L'objectif de notre projet de fin d'études est d'explorer l'utilisation de l'Analyse Logique de Données (ALD) pour la localisation de fautes dans les programmes informatiques avec de bonnes performances. Nous avons proposé deux approches basées sur l'ALD pour générer des ensembles d'items discriminant deux classes d'instructions : celles qui ont réussi (passed) et celles qui ont échoué (failed). Ces ensembles d'items devraient contenir l'erreur, permettant ainsi de la localiser. Nous avons adopté et implémenté,deux algorithmes : l'algorithme glouton hybride et l'algorithme d'extraction des traverses minimales (en anglais Minimal Hitting Set).Ces algorithmes ont permis d'extraire des ensembles d'instructions pertinents pour la localisation des fautes, avec des performances assez interessant .

Nous avons également introduit deux nouvelles mesures de suspicion, le XOR et N et !P, en plus de l'utilisation d'autres mesures issues de la littérature telles que Tarantula, Ochiai, Jaccard et GP13. Une comparaison entre ces différentes mesures à était effectuée afin de sélectionner la meilleure mesure qui permet d'optimiser les résultats.

Afin de prouver l'efficacité de nos approches, nous avons utilisé trois mesures de performance adaptées à ce domaine de détection de faute, il s'agit de l'exam score dans ses trois versions optimistes, pessimiste et le delta.

Les résultats obtenus sur le jeu de données nommée ISSTA13 (International Symposium on Software Testing and Analysis) de l'année 2013, composé de 10 programmes avec de multiples versions chacun, contenant une seule erreur à des emplacements de fautes variées, démontrent l'efficacité de nos approches proposées pour identifier les séquences d'instructions suspectes.

Cependant, l'étude révèle que la distinction entre les différentes mesures de suspicion n'est pas toujours claire, nécessitant des investigations plus approfondies. L'utilisation de l'intelligence artificielle explicable pour analyser les caractéristiques de chaque mesure et identifier leurs points forts et faibles pourrait apporter des éclaircissements précieux.

Par ailleurs, l'étude met en lumière les limites de nos approches, notamment en ce qui concerne les ressources matérielles importantes requises pour générer tous les itemsets discriminants, en particulier pour les programmes de grande taille. L'optimisation des algorithmes et l'exploration de techniques de parallélisation pourraient contribuer à réduire le temps de calcul.

Comme d'autres perspectives, nous suggérons d'étendre la recherche à la gestion de

scénarios multi-fautes, où plusieurs erreurs peuvent être présentes dans une même version du programme. De plus, l'utilisation de bases de données plus diversifiées et plus volumineuses permettrait une analyse des données plus approfondie. Une comparaison avec d'autres travaux permettrait également d'enrichir cette étude.

En conclusion, l'analyse de données logiques, enrichie par des nouvelles mesures de suspicion et couplée à des algorithmes permettant l'extraction d'itemsets discriminants tels que l'algorithme de glouton hybride et l'algorithme des Minimals Hitting Set, offre un potentiel prometteur pour la localisation de fautes dans les programmes informatiques. Des recherches futures s'avèrent nécessaires pour affiner cette approche, optimiser son efficacité et étendre son champ d'application, contribuant ainsi à l'amélioration de la qualité et de la fiabilité des logiciels.

Bibliography

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

[Alexe and Hammer, 2006] Alexe, G. and Hammer, P. L. (2006). Spanned patterns for the logical analysis of data. *Discrete Applied Mathematics*, 154(7):1039–1049.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

[Boros et al., 1997] Boros, E., Hammer, P. L., Ibaraki, T., and Kogan, A. (1997). Logical analysis of numerical data. *Mathematical programming*, 79(1):163–190.

[Boros et al., 2000] Boros, E., Hammer, P. L., Ibaraki, T., Kogan, A., Mayoraz, E., and Muchnik, I. (2000). An implementation of logical analysis of data. *IEEE Transactions on knowledge and Data Engineering*, 12(2):292–306.

[Briand et al., 2007] Briand, L. C., Labiche, Y., and Liu, X. (2007). Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 137–146. IEEE.

[Brun and Ernst, 2004] Brun, Y. and Ernst, M. D. (2004). Finding latent code errors via machine learning over program executions. In *Proceedings. 26th International Conference on Software Engineering*, pages 480–490. IEEE.

[Cellier et al., 2008] Cellier, P., Ducassé, M., Ferré, S., and Ridoux, O. (2008). Formal concept analysis enhances fault localization in software. In *International Conference on Formal Concept Analysis*, pages 273–288. Springer.

[Chambon, 2017] Chambon, A. (2017). *Caractérisation logique de données: application aux données biologiques*. PhD thesis, Université d'Angers.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351.

[DeMillo et al., 1978] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

[Dutta, 2024] Dutta, S. (2024). Localizing faults using verification technique. *Journal of Systems and Software*, 209:111897.

[F Delorme et al., 2024] F Delorme, L., Sais, D., Ing, S., and Jabbour (2024). Sélection de caractéristiques basée sur la lad pour des arbres de décision optimaux et autres classificateurs. *CRIL,CNRS Université d'Artois*.

[Gainer-Dewar and Vera-Licona, 2017] Gainer-Dewar, A. and Vera-Licona, P. (2017). The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100.

[Greenberg, 1998] Greenberg, H. J. (1998). Greedy algorithms for minimum spanning tree. *University of Colorado at Denver*.

[Gupta et al., 2005] Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272.

[Hammer, 1986] Hammer, P. L. (1986). Partially defined boolean functions and cause-effect relationships. In *Proceedings of the international conference on multi-attribute decision making via OR-based expert systems*. University of Passau.

[Jones et al., 2007] Jones, J. A., Bowring, J. F., and Harrold, M. J. (2007). Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26.

[Jones and Harrold, 2005] Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477.

[Laprie, 1992] Laprie, J.-C. (1992). Dependability: Basic concepts and terminology. In *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*, pages 3–245. Springer.

[Liblit et al., 2005] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. *Acm Sigplan Notices*, 40(6):15–26.

[Liu and Han, 2006] Liu, C. and Han, J. (2006). Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 46–56.

[Maamar et al., 2017] Maamar, M., Lazaar, N., Loudni, S., and Lebbah, Y. (2017). Fault localization using itemset mining under constraints. *Automated Software Engineering*, 24(2):341–368.

[Murakami and Uno, 2013] Murakami, K. and Uno, T. (2013). Efficient algorithms for dualizing large-scale hypergraphs. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–13. SIAM.

[Nessa et al., 2008] Nessa, S., Abedin, M., Wong, W. E., Khan, L., and Qi, Y. (2008). Software fault localization using n-gram analysis. In *Wireless Algorithms, Systems, and Applications: Third International Conference, WASA 2008, Dallas, TX, USA, October 26-28, 2008. Proceedings 3*, pages 548–559. Springer.

[Podgurski et al., 2003] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 465–475. IEEE.

[Renieres and Reiss, 2003] Renieres, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE.

[Tip, 1994] Tip, F. (1994). *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam.

[Valizadeh et al., 2024] Valizadeh, Z., Shams, M., and Dehghani, H. (2024). Eulerian-lagrangian dense discrete phase model (ddpm) of stenotic lad coronary arteries in comparison with single phase modeling. *Medical Engineering & Physics*, 128:104164.

[Wang and Roychoudhury, 2005] Wang, T. and Roychoudhury, A. (2005). Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351.

[Wang et al., 2022] Wang, T., Yu, H., Wang, K., and Su, X. (2022). Fault localization based on wide & deep learning model by mining software behavior. *Future Generation Computer Systems*, 127:309–319.

[Wong et al., 2008] Wong, E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE.

[Wong et al., 2010] Wong, W. E., Debroy, V., and Choi, B. (2010). A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208.

[Wong et al., 2011] Wong, W. E., Debroy, V., Golden, R., Xu, X., and Thuraisingham, B. (2011). Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169.

[Wong et al., 2016] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

[Wong and Qi, 2009] Wong, W. E. and Qi, Y. (2009). Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597.

[Yoo, 2012] Yoo, S. (2012). Evolving human competitive spectra-based fault localisation techniques. In *Search Based Software Engineering: 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings 4*, pages 244–258. Springer.

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10.

[Zhang et al., 2006] Zhang, X., Gupta, N., and Gupta, R. (2006). Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281.

Résumé

Le développement de logiciels est un processus complexe et itératif souvent sujet aux erreurs. Ces erreurs peuvent se manifester de diverses manières, allant de simples problèmes d'interface utilisateur à des défaillances critiques du système pouvant entraîner des pertes de données ou des comportements inattendus. Ceci est particulièrement dangereux, surtout s'il est question de déployer ces programmes sur du matériel. C'est pourquoi la détection et la correction précoces des fautes sont d'une importance capitale pour garantir la livraison de logiciels robustes, fiables et de haute qualité.

Dans ce mémoire de fin d'études, nous proposons nos approches basées sur l'Analyse Logique de Donne (ALD) dont l'objectif est de permettre aux programmeurs de localiser plus facilement les erreurs avec une bonne précision et ainsi de les corriger par la suite.

Nous avons développé quatre algorithmes, dont trois variantes de l'algorithme glouton (heuristique) et une méthode dite exacte. Avec glouton , nous extrayons d'abord plusieurs motifs suspects, puis nous les classons grâce à aux mesures de suspicion. Ceux ayant un score élevé contiennent plus souvent l'instruction fautive.

Nous avons testé nos approches sur une base de données open source nommée ISSTA13 (International Symposium on Software Testing and Analysis) de l'année 2013. Cette base de données contient 10 programmes présentant des fautes localisées à des emplacements distincts, ce qui en fait un outil idéal pour tester et évaluer nos algorithmes.

Pour évaluer les performances de nos approches proposées, nous avons utilisé des mesures d'évaluation des performances, il s'agit de l'exam score avec ses trois versions pessimistes et optimistes, ainsi que le delta Exam score. Les résultats illustrés sont satisfaisants.

Mots clés : Analyse Logique des Donnés (ALD), localisation des fautes, bese ISSTA13, l'algorithme Glouton (Greedy),motif , Mesures de suspicion.

Abstract

Software development is a complex and iterative process that is often prone to errors. These errors can manifest themselves in various ways, ranging from simple user interface issues to critical system failures that can lead to data loss or unexpected behavior. This is particularly dangerous, especially when it comes to deploying these programs on hardware. This is why early fault detection and correction are of paramount importance to ensure the delivery of robust, reliable, and high-quality software.

In this master's thesis, we propose our approaches based on Logical Data Analysis (LAD) whose objective is to allow programmers to more easily locate errors with good precision and thus correct them later.

We have developed four algorithms, including three variants of the Greedy algorithm and an exact method. With Greedy, we first extract several suspicious patterns, then we classify them using suspicion measures. Those with a high score more often contain the faulty instruction.

We tested our approaches on an open-source database called ISSTA13 (International Symposium on Software Testing and Analysis) . This database contains 10 programs with faults located in different locations, making it an ideal tool for testing and evaluating our algorithms.

To evaluate the performance of our proposed approaches, we used performance evaluation measures, namely the exam score with its three pessimistic and optimistic versions, as well as the delta Exam score. The illustrated results are satisfactory.

Keywords: Logical Data Analysis (LDA), fault localization, bese ISSTA13, the Greedy algorithm, pattern, Suspicion measures.

الملخص

تطوير البرمجيات هو عملية معقدة وتكرارية غالبًا ما تكون عرضة للأخطاء. يمكن أن تظهر هذه الأخطاء بطرق متنوعة، بدءًا من مشاكل بسيطة في واجهة المستخدم إلى فشل نظام حرج يمكن أن يؤدي إلى فقدان البيانات أو سلوك غير متوقع. هذا أمر خطير بشكل خاص، خاصة عند نشر هذه البرامج على الأجهزة. لذلك، فإن الكشف المبكر عن الأخطاء وتصحيحها هو أمر بالغ الأهمية لضمان تقديم برمجيات قوية و موثوقة وعالية الجودة.

في هذا البحث لنيل شهادة التخرج، نقترح نهجًا يعتمد على التحليل المنطقي للبيانات بهدف تمكين المبرمجين من تحديد الأخطاء بسهولة أكبر وبدقة عالية لتصحيحها فيما بعد.

قمنا بتطوير أربعة خوارزميات، ثلاث منها تعتبر تنويعات على الخوارزمية الطماعة (غريدي) وواحدة تعتمد على الطريقة الدقيقة. باستخدام غريدي ، نستخرج أولاً عدة أنماط مشبوهة، ثم نصنفها باستخدام مقاييس الاشتباه. الأنماط التي تحصل على درجة عالية تحتوي في الغالب على التعليمات الخاطئة.

لقد اختبرنا نهجنا على قاعدة بيانات مفتوحة المصدر تدعى ISSTA13 . تحتوي هذه القاعدة على ١٠ برامج بها أخطاء محددة في مواقع مختلفة، مما يجعلها أداة مثالية لاختبار وتقييم خوارزمياتنا.

لتقييم أداء مناهجنا المقترحة، استخدمنا مقاييس تقييم الأداء، وهي درجة الامتحان بإصداراتها الثلاثة المتشائمة والمتفائلة، بالإضافة إلى درجة امتحان دلتا. النتائج الموضحة مرضية.

الكلمات المفتاحية: تحليل البيانات المنطقية ALD ، موقع الخطأ، قاعدة ISSTA13 ، خوارزمية الجشع، النمط، مقاييس الشك.