UNIVERSITÉ PARIS NANTERRE

MÉMOIRE DE FIN D'ÉTUDES

MASTER 2 MIAGE

Recherche heuristique pour la détection de divergences de comportement entre programmes dans le cadre de l'apprentissage de la programmation

---

Présenté et soutenu par :

Valentin BOUQUET

6 septembre 2018

Sous la direction de : Monsieur Emmanuel HYON Maitre de conférences

Résumé

Dans le cadre de l'apprentissage de la programmation et de l'évaluation des programmes des apprenants nous sommes confrontés à la difficulté de générer automatiquement des critères d'évaluation. Écrire des tests pour s'assurer de la conformité d'un programme avec un comportement souhaité demande du temps et une certaine expertise. A cause de ces contraintes, la qualité des tests peut être négligée et il préférable de disposer d'outils qui permettent de détecter automatiquement les programmes dont le comportement est divergent. Dans ce cadre là, nous présentons les grandes techniques du test logiciel et plus spécifiquement de la génération automatique de jeux de tests ainsi qu'une présentation des heuristiques de recherches simples et avec apprentissage par renforcement dans le but de palier au limitations des méthodes existantes, plus spécifiquement l'exécution symbolique. Enfin nous présentons une ébauche de méthode pour résoudre notre problématique avec les techniques présentées précédemment.

Mots clés : Heuristique de recherche, Apprentissage, Test logiciel, Génie logiciel, Exécution symbolique, Automatisation

Rermerciements

Cette année est l'aboutissement de 5 années d'études universitaires, je tiens donc à saluer tous les enseignants et enseignants-chercheurs que j'ai pu côtoyer durant toutes ces années ainsi que tous les étudiants, de la licence au master de la *MIASH* et de la *MIAGE* de l'Université de Paris-Nanterre.

Je pense aussi aux opportunités qui se sont présentées à moi afin de poursuivre mes études sereinement dans le supérieure. Il ne fait aucun doute que le climat universitaire français est généreux et que sans lui il m'aurait été impossible de continuer comme je l'ai fait. J'espère que les générations futures seront tout aussi à mène d'en bénéficier et de savourer cette chance. En espérant que les entraves à ce système s'essoufflent et laissent place à un avenir plus prometteur.

Je remercie Monsieur Emmanuel Hyon pour son encadrement, son écoute et pour sa bienveillance envers ses étudiants ainsi que pour ses conseils qu'il a su me prodiguer, y compris pour la rédaction de ce mémoire.

Je remercie chaleureusement Monsieur François Delbot et Monsieur Jean-François Pradat-Peyre pour tout le soutien qu'ils m'ont porté et de m'avoir permis de travailler dans d'excellentes conditions. Je les remercie vivement pour tout les conseils, avis et discussions partagés qui m'ont ouvert vers de nouvelles horizons. Sans leurs soutiens, je n'aurais jamais une seconde pu imaginer poursuivre un travail de recherche. J'espère que ces récents espoirs seront trouver satisfaction.

Table des matières

Introduction **1**

1 Test logiciel et automatisation **3**
1.1 Introduction . . . 3
1.2 Génération de tests . . . 4
1.2.1 Exécution symbolique . . . 4
1.2.2 Model-Based Testing . . . 7
1.2.3 Search-Based Testing . . . 8
1.3 Détection de divergences entre versions d'un logiciel . . . 10
1.3.1 Méthode utilisée . . . 10
1.3.2 Limitations . . . 11

2 Heuristique de recherche **13**
2.1 Définitions et intérêt . . . 13
2.1.1 Problème du voyageur de commerce (TSP) . . . 13
2.1.2 Un algorithme glouton . . . 14
2.2 Modélisation . . . 16
2.2.1 Définitions générales . . . 16
2.2.2 ET-OU graphe . . . 17
2.2.3 Représentation d'état . . . 19

3 Procédures de recherches **21**
3.1 Procédures de recherches simples . . . 21
3.1.1 Recherche non informée . . . 21
Parcours en profondeur (DFS) . . . 21
Parcours en largeur (BFS) . . . 22
3.1.2 Recherche informée . . . 23
Best-First search (le meilleur d'abord) . . . 23
Algorithmes best-first spécialisés . . . 24
3.2 Procédures de recherches avec apprentissage . . . 25
3.2.1 Monte Carlo Tree Search (MCTS) . . . 25

4 Application **27**
4.1 Contexte . . . 27
4.1.1 Apprentissage de la programmation et tests . . . 27
4.1.2 Limitations des méthodes existantes . . . 27
4.1.3 Objectifs . . . 28
4.2 Méthode proposée . . . 28
4.2.1 Présentation . . . 28
4.2.2 Problématiques restantes . . . 31

Conclusion 33

Bibliographie 35

Table des figures

1.1 Exemple des chemins d'exécutions possibles récoltés grâce à l'exécution symbolique d'un programme C. 5
1.2 Publications dans le domaine du *Search-Based Software Testing* - source : *Search-Based Software Testing : Past, Present and Future*[9] 9
1.3 Exemple d'un branchemement de l'exécution symbolique tenant compte de la nouvelle version du programme - source : *Article shadow[2]* 10

2.1 Un chemin possible pour résoudre le problème du voyageur de commerce. 13
2.2 Représentation d'un graphe à 7 sommets et 7 arêtes avec un chemin dessiné en bleu entre le sommet A et le sommet H : {A, D, F,G, H}. 17
2.3 Un arbre uniforme de profondeur 2 à 7 sommets et 6 arêtes . . 17
2.4 Respectivement les arcs ET et OU du graphe de réduction de problème. 18
2.5 Problème de la pièce contrefaite représenté avec un ET-OU graphe[11] 19
2.6 Représentation sous forme d'état de l'exécution symbolique d'un programme 20

3.1 Ordre dans lequel les sommets sont visités avec un parcours en profondeur (gauche) et sa variante (droite) : le backtracking. 22
3.2 Ordre dans lequel les sommets sont visités avec un parcours en largeur (gauche) et sa variante (droite) : la procédure du coût uniforme. 23
3.3 Une itération du MCTS - source : *A Survey of Monte Carlo Tree Search Methods*[13] 26

Introduction

Motivations

C.A.T. est une plateforme pédagogique en ligne dont l'objectif est de proposer des outils d'aide à la création de supports de cours et d'exercices dans le cadre de l'apprentissage de la programmation. Pour faciliter l'encadrement et apporter une certaine autonomie aux étudiants, un des enjeux primaire rencontré par cette plateforme est celui de l'automatisation de la correction des programmes des apprenants et des retours d'évaluations. Évaluer le code source informatique d'un étudiant est une tâche laborieuse et délicate. Le code informatique demande une certaine rigueur et la relecture du code d'une personne peu expérimentée est une tâche difficile et fatigante. Un programme a priori juste peut échouer à cause d'erreurs de syntaxes jugées minimes.

Actuellement un encadrent doit lui même, à l'aide des outils mis à disposition par *C.A.T.*, définir les tests que devront passer les programmes des étudiants pour être jugés correct. Pour cela, nous avons fait le choix de proposer des outils favorisant la comparaison entre le programme de la correction, fournit par l'encadrent et le programme source de l'étudiant. Il s'agit donc de tester la conformité du programme correction avec le programme donné par un apprenant qui aura été réalisé à l'aide des spécifications des éléments attendus dans l'énoncé de l'exercice.

Bien que l'automatisation de cette correction permette de faciliter l'analyse en factorisant la correction sur les programmes des étudiants (création d'un jeu de tests pour n programmes), cette automatisation nécessite toujours la participation active de l'encadrent lors de l'écriture des tests. L'écriture des tests n'est pas aisée puisqu'elle demande une certaine expertise technique allant au delà du langage de programmation. De plus, l'encadrent peut, par faciliter ou simplement par manque de temps, négliger la qualité de ses tests et omettra de couvrir certains cas importants. Ce seront donc potentiellement des programmes que les jeux de tests auront validé qui pourraient être erronés dans certaines configurations.

L'objectif fixé est de favoriser la mise à disposition automatisée de jeux de tests pour faciliter le travail des évaluateurs. Proposer des valeurs qui soient intéressantes pour couvrir les erreurs les plus fréquentes est une tâche difficile. Des outils d'analyse statique et dynamique permettent aujourd'hui de déchiffrer en partie le comportement des programmes mais ces techniques sont parfois limitées à cause de la complexité des programmes et des contraintes des machines et des outils. Il existe donc un intérêt à disposer d'outils qui permettent de détecter automatiquement des divergences de comportement entre programmes et qui soient exécutables en un temps raisonnable. Le but

fixé n'est pas de détecter forcément tous les éventuels cas de divergences mais de se servir d'heuristiques pour parcourir le comportement des programmes en un temps limité tout en classifiant les éléments qui sont à l'origine d'éventuelles divergences de comportements. Nous espérons pouvoir tirer parti de cette classification pour limiter les chemins d'exécution à parcourir des autres programmes. Nous devons donc trouver des solutions pour détecter au moins un chemin d'exécution qui donne un comportement divergeant pour utiliser les informations des contraintes sur ce chemin pour faciliter les parcours des chemins d'exécutions possibles des autres programmes.

Objectifs du mémoire

Dans ce mémoire nous présentons dans un premier temps la thématique du test logiciel et plus spécifiquement des techniques de génération automatique de tests (exécution symbolique, model-checking, méthodes aléatoire) ainsi que leurs limitations et une méthode proposée pour détecter des divergences entre deux versions d'un même logiciel (cas le plus proche de ce que nous souhaitons faire). Puis nous introduirons la notion d'heuristique et les différentes modélisations possibles et compléterons sur les procédures de recherche simples ainsi que sur celles avec apprentissage comme le Monte Carlo Tree Search (MCTS) afin d'y voir plus clair sur les méthodes utilisées pour diminuer la complexité. Enfin nous établirons les liens possibles entre les méthodes de recherche heuristique avec apprentissage par renforcement et l'exécution symbolique pour faciliter la détection de divergence de programmes dans le cadre de l'apprentissage de la programmation.

Chapitre 1

Test logiciel et automatisation

1.1 Introduction

« Tester revient à confronter par des moyens statiques (analyse de code, revue, etc.) ou par des moyens dynamiques (exécution avec des valeurs particulières) les spécifications du logiciel, c'est à dire ce qu'il doit faire et éventuellement sous quelles contraintes (temps, utilisation de la mémoire, etc.), à sa réalisation, c'est à dire de quelle façon il répond au besoin exprimé en enchainant différentes actions élémentaires. »[4]

Le test logiciel est une part importante du cycle de développement d'une application et devient vite indispensable pour pérenniser les projets puisqu'elle permet à la fois de s'assurer du comportement d'un logiciel mais aussi de faciliter la maintenance en écartant tous cas de régression lors de l'ajout de nouvelles fonctionnalités ou de simples patchs correctifs.

L'importance de cette activité dépend fortement du domaine de l'application. Par exemple quand la sécurité des données d'une société, d'un chef d'état ou des vies humaines peuvent être impactés par l'exécution d'un programme comme c'est le cas dans les logiciels embarqués en aéronautique, elle prend une place prépondérante et on ira jusqu'à utiliser des techniques de vérification pour s'assurer à un taux proche de 100% que l'exécution du logiciel correspond aux spécifications définies. Une telle assurance est parfois fondamentale puisque une erreur lors de l'écriture du logiciel peut causer un *bug* majeur lors son exécution et être particulièrement couteuse. Un des cas les plus célèbres est celui-ci associé au programme du décollage de la fusée Ariane 5[8] qui eu pour cause l'explosion de celle ci en plein vol après seulement quelques secondes de décollage. Erreur causée par le plantage du système de guidage inertiel principal à cause d'un simple dépassement d'entier, qui aurait donc pu être corrigé en testant les bornes maximales des entiers.

Cette activité à tout de même de nombreux inconvénients puisqu'elle demande du temps pour être réalisée correctement mais aussi une certaine expertise et est plutôt exigeante intellectuellement. Il ne s'agit pas de tester le programme au hasard en espérant trouver des erreurs mais il faut parfois penser plus loin que le comportement général du programme et donc analyser aussi le comportement des fonctions, des composants et de leurs intégrations avec d'autres logiciels ou d'autres environnements.

Le test logiciel est donc couteux et on lui attribue souvent à lui seul 50% à 75% du coût total de développement[14] pour que les logiciels produits

soient un minimum fiables. C'est donc impératif pour l'industrie de réduire le coût et d'améliorer son efficacité tout en automatisant son processus.

Il est aussi indispensable au développement logiciel et c'est une des parties fondamental des disciplines du génie logiciel. Aucune surprise que ce domaine et plus spécifiquement que la génération automatique de tests est fait l'objet d'autant de recherche ces dernières années et de nombreux outils et techniques ont émergés[10]. En même temps, les logiciels se sont complexifiés pour tenir compte des nouvelles techniques de programmation facilitant le développement, la maintenance, les modifications mais aussi devant supporter le bagage informatique des années précédentes. Cette hétérogénéité peut devenir problématique, puisque les programmes peuvent avoir des supports différents, que ce soit le *smartphone*, l'ordinateur de bureau, les consoles de jeux ou même les objets connectés (*IOT* : Internet Of Things) qui peuvent interagir avec l'environnement physique tout en communicant avec d'autres machines sur le réseau.

1.2 Génération de tests

La génération automatique de tests[10] est un domaine déjà bien développé qui permet aujourd'hui grâce à différentes analyses à partir du code comme celle de l'arbre syntaxique abstrait, du graphe de flot de contrôle (CFG) et d'application de méthodes comme l'exécution symbolique, concolique ou de model-checking, d'extraire des propriétés parfois abstraites du code pour identifier des jeux de tests pertinent pour s'assurer de la qualité du programme[15]. Deux analyses majeures sont utilisées qui sont l'analyse statique, qui couvrent les méthodes permettant d'obtenir des informations d'un programme à partir de son code source sans l'exécuter et l'analyse dynamique qui couvrent les méthodes utilisées pour observer le comportement d'un programme à l'exécution.

1.2.1 Exécution symbolique

Définitions

L'exécution symbolique[7] est une technique de type boîte blanche (on peut observer la structure interne du programme) qui permet d'analyser le comportement d'un programme en remplaçant les données par des symboles. Au lieu d'exécuter le programme et de suivre l'évolution des valeurs des variables concrètes en mémoire et comment elles influent sur le comportement du programme, on construit un arbre des chemins d'exécutions possibles à partir de valeurs symboliques en entrée et des contraintes récoltées sur les différents chemins. Au final, les valeurs de sorties calculées par le programme seront exprimées comme fonction des valeurs symboliques d'entrée.

Dans le domaine du test logiciel, l'exécution symbolique est utilisée pour générer des jeux de tests pour couvrir les branches d'exécutions possibles du

programme ou pour s'assurer que certains chemins d'exécutions sont irréalisables sous certaines conditions. Un chemin d'exécution possible (exemple figure : 1.1) est une séquence de condition qui sont soient « vraie » soit « fausse » exprimée en fonction des contraintes d'entrées et celles récoltées tout au long du chemin. Tous les chemins d'exécution du programme peuvent être représentés en utilisant une structure d'arbre.

On appelle la contrainte de chemin (Path constraint : PC) une formule booléenne qui représente l'équation de toutes les contraintes que doivent assurer les valeurs symboliques d'entrées pour suivre un chemin d'exécution. Par exemple, la figure 1.1 montre en bas à gauche un exemple de contrainte de chemin ($[PC] : x < 0 \bigwedge x > -10$) signifiant que la variable symbolique x en entrée doit satisfaire ces contraintes pour suivre ce chemin d'exécution et on peut donc voir que chaque chemin d'exécution possède ses propres contraintes.

```
int exemple(int x)
{
  if (x < 0) {
    x = x * 3.14;
    if (x > -10) {
      // ...
    }
    else {
      // ...
    }
  }
  else {
    if (x == 1) {
      // ...
    }
    else {
      // ...
    }
  }
  // ...
}
```

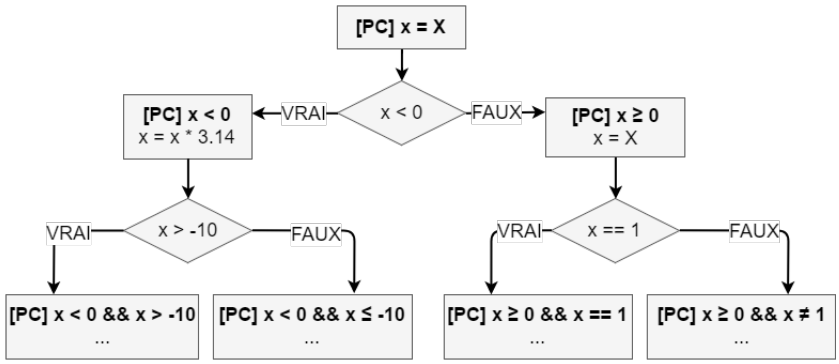

FIGURE 1.1 – Exemple des chemins d'exécutions possibles récoltés grâce à l'exécution symbolique d'un programme C.

Limitations

Bien que la technique d'exécution symbolique a été proposée dans les années 70[7], cette technique n'a reçu l'attention des chercheurs que récemment pour deux raisons. Premièrement l'application de l'exécution symbolique sur des programmes conséquents en application réelle requiert de résoudre des équations complexes avec beaucoup de contraintes. C'est là où les récentes avancées des solveurs de contraintes ont été bénéfiques. Un solveur de contrainte permet de résoudre des équations de satisfaction de contraintes. Il est donc capable d'évaluer la faisabilité d'un ensemble de contraintes donné

et permet donc de déterminer si un chemin donné par l'exécution symbolique est faisable ou non (*SAT* ou *UNSAT*). Ces outils sont donc utilisés conjointement à l'exécution symbolique pour limiter l'exécution sur les chemins atteignables. Deuxièmement, l'exécution symbolique est une technique extrêmement couteuse en ressource, bien plus que les méthodes classiques d'analyse statique. Ce coût est associé à la multiplicité des chemins d'exécution possibles dans des programmes complexes et à la complexité des contraintes des chemins qui augmentent au fur et à mesure du parcours. Heureusement, les puissances de calculs actuelles ont largement augmentées et permettent de réaliser cette analyse sous certaines conditions.

Même si l'exécution symbolique semble être théoriquement capable d'analyser en profondeur le comportement d'un programme, cette technique souffre encore de problèmes majeurs quand elle est confronté à de réels programmes :

— Le premier est l'explosion des chemins, c'est à dire que le nombre de chemins d'exécutions possible évolue exponentiellement. Il est donc impératif de trouver des méthodes pour réduire le nombre de chemins à évaluer.

— Le deuxième est la divergence de chemins ou *path divergences*. On parle de chemin divergent quand le comportement analysé par l'exécution symbolique est différent du comportement concret à l'exécution. Un tel cas peut arriver plus fréquemment qu'on ne le pense puisque les programmes s'exécutent sur des environnements complexes communicant avec plusieurs composants. L'exécution symbolique ne fait qu'isoler le comportement du programme et il est difficile de tenir compte de tous les éléments pouvant affecter le comportement du programme.

— Le troisième est la complexité des contraintes qui peuvent parfois être trop importantes pour être résolus par un solveur de contraintes. Des contraintes impliquant des opérations de multiplication et de division ou d'autres opérations non linéaires ont tendances à faire rapidement exploser la complexité des contraintes. Si les contraintes de chemins sont indécidables, alors les chemins possiblement couverts par l'exécution symbolique sont limités.

Techniques pour contourner les limitations

Bien que ces problématiques limitent fortement l'utilisation des techniques d'exécution symbolique sur de réelles applications, il existe des méthodes qui permettent en partie de contourner ces limitations. Par exemple, pour contourner le problème d'explosion des chemins, des techniques permettent de guider l'exploration en ce limitant aux chemins amenant à un certain objectif, par exemple celui de couvrir une portion du programme. Pour limiter la complexité grandissante des contraintes du problème, il est possible de joindre l'exécution symbolique du programme à son exécution concrète, c'est l'exécution concolique. Dans ce cas de figure, le programme est exécuté sur

des valeurs concrètes ainsi que sur des valeurs symboliques ce qui permet, si des contraintes deviennent trop complexes, de les simplifier en utilisant les valeurs concrètes du programme. Au final, on se sert de l'exécution du programme pour construire les chemins d'exécution et de l'exécution symbolique pour limiter le nombre de chemins à exécuter réellement.

1.2.2 Model-Based Testing

Définitions

Model-Based Testing (MBT) est une méthode semi formelle qui utilise des modèles du système testé (*System Under Test : SUT*) pour générer des cas de tests. Les modèles peuvent représenter des abstractions du comportement du SUT et ils pourront être utilisés pour comparer le comportement théorique modélisé au comportement réel du SUT. De manière générale, le SUT est modélisé comme une boite noire, c'est à dire que son comportement interne n'est pas analysé. Les modèles décrivent alors des séquences d'entrées et de sorties possibles. Un des gros avantages de ce type de méthode est qu'elle permet d'abstraire le comportement souhaité du logiciel du code et un modèle pourrait donc définir le comportement souhaité d'un programme, qu'il soit écrit en C ou en Java par exemple.

Limitations

Dans notre démarche, nous souhaitons comparer le comportement de programmes dont l'exécution devrait être identique. Nous avons donc une approche qui est orientée à partir du code, c'est à dire que l'objectif des programmes dépend du code source de la programmation qui décrit le comportement souhaité. Dans une approche à partir de modèles, l'objectif est souvent inverse. Il est plus facile de décrire les modèles et ensuite de générer une partie du code et des tests. Si nous avions décidé d'écrire les exercices de programmation sous forme de modèles, nous aurions pu estimer plus facilement les bénéfices d'une méthode de modélisation. Nous savions qu'une telle approche nécessiterait une compétence technique supplémentaire de l'évaluateur qui devrait être en mesure de décrire le comportement du programme souhaité sous forme de modèles et c'est pour cela que cette méthode n'a pas été privilégiée. De plus, la quantité d'information modélisée est souvent limitée puisqu'il s'agit de définir principalement le comportement attendu en fonction des entrées et des sorties. Avec une approche type boîte blanche, nous pouvons observer le comportement interne d'un programme et potentiellement découvrir des éléments internes qui favorise l'émergence d'une divergence de comportement entre deux programmes.

1.2.3 Search-Based Testing

Définitions

Search-based software testing[9] est un sous domaine de *Search-based software engineering*[5] qui regroupe l'application des heuristiques de recherche pour résoudre des problèmes du génie logiciel. *Search-based testing*, lui, est le domaine qui regroupe les techniques de recherche heuristique à des fins de générations de tests. Pour cela, il s'agit de transformer le problème de génération de tests en un problème d'optimisation qui peut être résolu avec ces techniques.

Les techniques de recherche heuristiques sont des alternatives pour attaquer les problèmes où l'espace de recherche est trop important. Cette situation est fréquemment rencontrée lorsqu'il s'agit d'établir l'ensemble des chemins d'exécutions possibles d'un programme pour déterminer des jeux de tests pertinents. L'objectif n'est donc pas de couvrir toutes les branches possibles, mais de définir une fonction objectif qui guidera automatiquement la recherche pour atteindre un but fixé le plus rapidement possible. C'est d'ailleurs le point crucial de cette méthode, puisque c'est cette fonction objectif qui permettra à l'algorithme de déterminer de bonnes solutions dans un temps raisonnable. Elle peut donc se montrer efficace pour générer des jeux de tests pertinents et pour optimiser le processus de génération de tests.

La forme la plus simple d'un algorithme d'optimisation - bien que très peu efficace - est une recherche aléatoire. Ici, les jeux de tests sont générés aléatoirement jusqu'à ce que l'objectif soit atteint. Un exemple d'objectif simple peut être le pourcentage de couverture du programme à atteindre. Malheureusement cette méthode est très limitée puisque plus l'espace des valeurs possibles est grand moins les valeurs générées aléatoirement auront une chance d'être suffisamment pertinentes.

D'autres techniques sont plus fréquemment utilisées comme celle du *Hill Climbing* qui, appliquée à l'exécution symbolique, favorise la sélection d'optimum locaux. L'algorithme favorise lors de son parcours les chemins les plus prometteurs, c'est à dire que depuis chaque branchement de l'exécution, il sélectionnera le chemin le plus prometteur (par exemple se rapprochant le plus de la solution). Comme un chapitre entier est réservé aux heuristiques, nous ne décrirons pas plus en détail les méthodes de parcours utilisées puisqu'elles sont quasiment identiques.

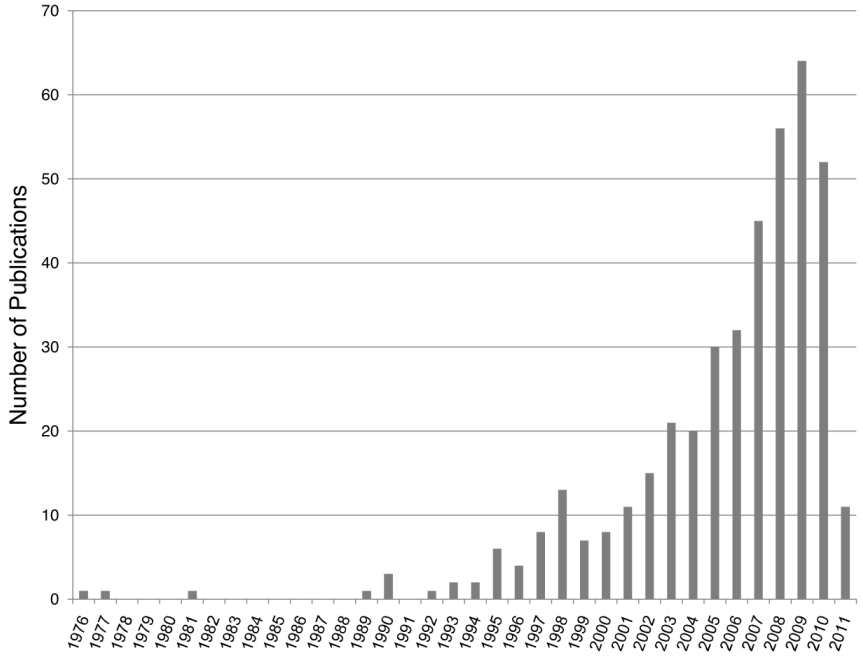

FIGURE 1.2 – Publications dans le domaine du *Search-Based Software Testing* - source : *Search-Based Software Testing : Past, Present and Future*[9]

Limitations

Le succès de ces méthodes dépend majoritairement de la qualité de la fonction objectif fixée qui doit permettre de satisfaire un ensemble de critères comme par exemple un pourcentage de couverture des branches à atteindre. De plus, il faut aussi que les outils disponibles puissent tenir compte des critères sélectionnés (le critère peut être indéterminable statiquement ou dynamiquement sur une portion de code) et que l'évaluation de ces critères lors de l'analyse ne soit pas trop longue.

Une fois l'objectif défini, il existe de nombreuses heuristiques de recherche possibles à appliquer. Il faut donc sélectionner la bonne méthode de recherche qui soit apporte un résultat de meilleur qualité, soit le fait en un temps plus raisonnable ou bien les deux à la fois. Au final, tout le succès de ces méthodes dépend de la définition de la fonction objectif qui doit avoir des critères adéquats et déterminables selon la méthode utilisée mais aussi d'une recherche heuristique adaptée au problème qui fasse un compromis entre la qualité de la solution obtenue et le temps d'exécution.

C'est d'ailleurs une limitation importante de ces techniques de recherche puisqu'elles ne sont majoritairement appliquées que pour générer des jeux de tests afin de couvrir la plus grand portion du code possible (ou des régions particulière du code) sans se soucier du comportement de l'application à tester en fonction des valeurs d'entrées. Heureusement, les méthodes heuristiques permettent d'adapter l'objectif pour qu'il soit pluriel. A l'avenir les défis sont donc de définir des objectifs qui tiennent comptent de multi-critères qui sont déterminables lors de l'analyse et qui permettent de tenir

compte du comportement du programme pour générer des jeux de tests de qualité.

1.3 Détection de divergences entre versions d'un logiciel

1.3.1 Méthode utilisée

Pour la détection de divergences de comportements entre deux versions d'un même logiciel, il existe des solutions qui permettent de limiter la combinatoire et notamment pour l'exécution symbolique. Une de ces méthodes proposées est présentée dans l'article *Shadow of a doubt : Testing for divergences Between Software Versions*[2] (en référence au célèbre film Alfred Hitchcock : *L'ombre d'un doute*).

Le postulat de départ est qu'il serait moins coûteux de comparer deux versions d'un même logiciel uniquement aux endroits où le code à été modifié. Au lieu de se baser sur l'analyse complète des deux arbres d'exécution symbolique du programme, nous partirions donc des sous-arbres de ces ensembles, là ou la divergence de code est présente, pour établir l'analyse. Pour cela, il est proposé de fondre les deux versions du programme en tenant compte des modifications effectuées et ceci grâce à quelques annotations qui permettent de générer des branches d'exécutions séparées là où le code diverge.

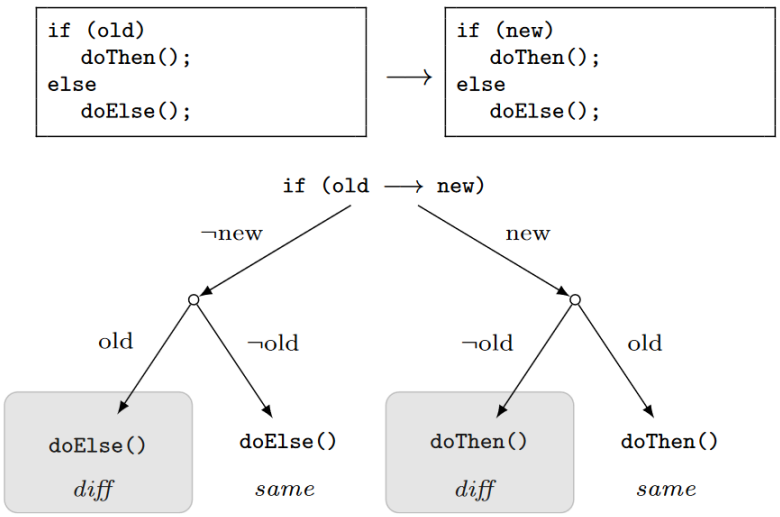

FIGURE 1.3 – Exemple d'un branchement de l'exécution symbolique tenant compte de la nouvelle version du programme - source : *Article shadow[2]*

1.3.2 Limitations

Même si cette approche paraît particulièrement intéressante, nous doutons de son intérêt dans notre cas où nous ne disposons pas de deux versions d'un même programme mais de n programmes implémentant tous une même spécification (description en langue naturel ou par des formules mathématiques de ce qui doit être produit). Potentiellement, ces programmes pourraient être intégralement différents dans leur structure bien que leurs comportements à l'exécution seraient identiques. Il est donc plus difficile de bénéficier de la proximité des programmes pour limiter la combinatoire de l'exécution symbolique.

Conclusions :

Les approches que nous avons vu précédemment sont souvent limitées[6] de par la complexité grandissante des programmes qui confrontent ces méthodes à l'explosion du nombre de chemins, du nombre d'états ou de contraintes possibles. Pour représenter ces états et chemins, ces méthodes reposent sur des modèles d'arbres qui ne peuvent être parcourus en intégralité à cause de leur taille. Les algorithmes de parcours actuellement utilisés reposent majoritairement sur des stratégies simples comme le parcours en profondeur qui est connu pour être particulièrement couteux mais qui a le bénéfice de trouver au moins un chemin d'exécution possible.

Ces dernières années, grâce à la maturation des recherches sur les différentes stratégies et heuristiques de parcours de graphe (notamment celles avec apprentissage) et des applications récentes sur des problèmes concrets, comme celui du jeu de go où l'ordinateur *AlphaGo*[1] qui grâce à une implémentation de l'heuristique du Monte Carlo Tree Search a réussi à battre le champion du monde actuel, ces méthodes ce sont popularisées.

C'est pourquoi, nous pensons que des méthodes de Search-based Testing combiner à des méthodes d'exécution symboliques pourraient permettre de contourner nos contraintes. Nous espérons pouvoir utiliser ou simplement nous inspirer des heuristiques avec apprentissages pour diminuer l'espace de recherche plus nous analysons de programme.

Chapitre 2

Heuristique de recherche

2.1 Définitions et intérêt

Les heuristiques sont des critères, méthodes ou principes utilisées pour sélectionner une solution efficace parmi un ensemble possible afin d'atteindre un ou plusieurs objectifs fixés[11]. Elles ne sont d'ailleurs pas toujours justes ou fiables dans toutes les situations et peuvent donc être hasardeuses.

Pour une grande majorité des problèmes complexes, déterminer une solution exacte nécessite d'évaluer un immense ensemble de choix. Le temps requis pour trouver cette solution peut être trop important et il est nécessaire parfois de faire des compromis pour obtenir une solution efficace en un temps raisonnable en utilisant une heuristique.

Elles sont particulièrement utilisées pour répondre aux problèmes dits NP-complet. Ce sont les problèmes pour lesquels tous les algorithmes connus requièrent un temps exponentiel en pire cas pour être résolus. Un des exemples les plus connus et fréquemment enseigné en cours d'algorithmique est le problème du voyageur de commerce.

2.1.1 Problème du voyageur de commerce (TSP)

Soit un ensemble de n villes réparties et un voyageur souhaitant toutes les parcourir une et une seule fois puis retourner à la ville d'où il est parti et ce en parcourant au total la distance la plus petite possible.

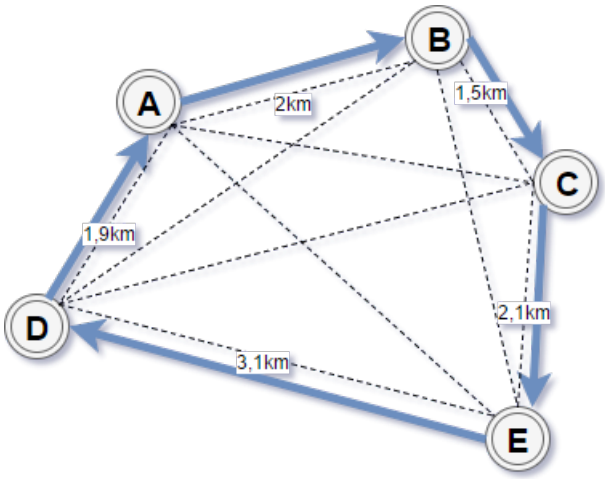

FIGURE 2.1 – Un chemin possible pour résoudre le problème du voyageur de commerce.

Il existe au total $n!$ chemins possibles soit dans notre cas 120. La ville de départ n'ayant aucune influence sur la longueur totale parcourue on peut donc réduire l'ensemble à $(n-1)!$ soit 24 chemins. Enfin chaque chemin pouvant être parcouru dans les deux sens sans impacter la distance, on peut donc réduire l'ensemble final à $\frac{1}{2}(n-1)!$ soit 12 chemins.

Villes	Chemins (solutions)
5	12
10	181400
15	43 589 145 600
20	60 822 550 204 416 000

TABLE 2.1 – Évolution de l'ensemble des chemins en fonction du nombre de villes pour la résolution du problème du voyageur de commerce

Le tableau ci-dessus rend compte de la rapidité à laquelle le nombre de chemins à évaluer grandit en fonction du nombre de villes. On parle d'*explosion combinatoire* : c'est le fait qu'un problème se complexifie grandement lorsque le nombre de données à considérer augmente légèrement et peut rendre sa solution incalculable en un temps restreint (longévité humaine par exemple).

2.1.2 Un algorithme glouton

Pour trouver une solution (pas forcément la meilleure) au problème du voyageur de commerce, nous pouvons utiliser une heuristique simple en utilisant un algorithme glouton. Un tel algorithme repose sur le fait de dérouler les données de manière itérative en sélectionnant à chaque étape un optimum local. Ceci a pour effet de grandement diminuer le nombre de données à considérer et donc de répondre en partie à l'explosion combinatoire.

L'algorithme 1 présente une heuristique simple pour répondre au problème d'explosion des chemins en diminuant le nombre de données à évaluer. Depuis la ville de départ u (choisit aléatoirement par exemple), il s'agit de sélectionner la ville la plus proche parmi les $n-1$ villes restantes. Puis de manière itérative, nous sélectionnons la prochaine ville la plus proche depuis la dernière ville sélectionnée et ceci jusqu'à ce que toutes les villes soient sélectionnées. A la première itération nous avons donc $n-1$ distances à évaluer puis nous en aurons $n-2$ à la deuxième. Au final cet algorithme doit évaluer $\frac{n(n-1)}{2}$ distances.

Algorithm 1 Problème du voyageur - un algorithme glouton

```
function DETERMINER_CHEMIN(villes, n)
    P := LISTE_VIDE
    choisir un sommet u dans villes
    villes := villes − u
    P := P ∪ u
    while |P| ≠ n do
        d := +∞
        for v in villes do                ▷ Évaluation de la ville la plus proche
            if distance(u, v) < d then
                d := distance(c, v)
                u' := v
            end if
        end for
        u := u'
        P := P ∪ u
        villes := villes − u
    end while
    return P
end function
```

Ceci montre un exemple simple d'utilisation d'une heuristique qui repose sur la découpe du problème en sous-problèmes pour réduire l'ensemble des données du domaine. L'inconvénient est qu'une telle méthode ne donne pas de garantie de résultât car le chemin le plus court possible n'est retourné que dans le meilleur des cas. La sélection d'une heuristique pour répondre à un problème réside dans le compromis entre le temps requis pour obtenir une solution et la qualité de la solution retournée c'est à dire sa proximité avec la meilleure solution possible. Ces deux critères peuvent être évalués en moyenne, dans le pire cas possible, dans le meilleur cas possible ou bien les trois à la fois.

Villes	Chemins	Chemins (algorithme glouton)
5	12	10
10	18 1400	45
15	43 589 145 600	105
20	60 822 550 204 416 000	190

TABLE 2.2 – Comparaison du nombre de chemins évalués pour la résolution du problème du voyageur de commerce avec un algorithme glouton (heuristique).

Les heuristiques sont intéressantes en dehors du domaine théorique car la majorité des problèmes pratiques ne nécessite pas d'établir la solution la plus optimale. On préférera trouver un équilibre entre la qualité de la solution obtenue et le coût pour trouver une telle solution qui est un critère non

négligeable si l'on prend compte du contexte économique. On parle alors de problème de *semi-optimisation* et plus particulièrement d'optimisation proche lorsque qu'il s'agit de trouver une solution dans un intervalle de coût définie ou de problème d'optimisation approximatif lorsqu'il s'agit de se rapprocher de l'optimum avec une probabilité importante.

2.2 Modélisation

Pour établir une bonne heuristique et évaluer sa capacité à produire des solutions en un temps défini sur un problème donné, il convient de correctement représenter ce problème. De nombreux problèmes peuvent être formulés comme problème de satisfaction de contraintes - où l'on cherche des états ou des objets satisfaisant un certain nombre de critères - et d'optimisation de tâches. De plus, une heuristique doit pouvoir être automatisée et donc être résolu à l'aide des machines actuels.

Puisque toutes les recherches de solutions à un problème peuvent se résumer à la tâche de construire un object avec les caractéristiques données, les besoins[11] pour la résolution avec un ordinateur sont les suivants :

1. Une structure de symbole appelée code ou base de données représentant les sous-ensembles des solutions potentielles.
2. Un ensemble d'opérations ou des règles de production qui modifient les symboles de la base de données pour produire un sous-ensemble de solutions plus fins ou précis.
3. Une procédure de recherche ou stratégie de contrôle qui décide quelles opérations sont à appliquer sur la base de données.

2.2.1 Définitions générales

Les différentes façons de représenter nos problèmes repose majoritairement sur des modèles de graphe. Vous pouvez passer à la section suivante si vous possédez déjà des connaissances de base de théorie des graphes sinon nous décrivons brièvement les notions importantes ici :

Graphe :
Un graphe est composée d'un ensemble de **nœuds** ou **sommets** reliés par des **arcs** ou **arêtes** pouvant être associées à des valeurs (par exemple la distance entre deux sommets) ou bien être dirigé (donnant la direction, on va d'un nœud à l'autre). Dans notre cas, nos graphes auront toujours un nœud de départ appelé **nœud racine**. L'ensemble des nœuds est le plus souvent noté V et on note E pour l'ensemble des arêtes du graphe. Un graphe est mathématiquement représenté de cette façon : $G = (V, E)$. Le **degré** d'un sommet est le nombre d'arêtes de celui-ci.

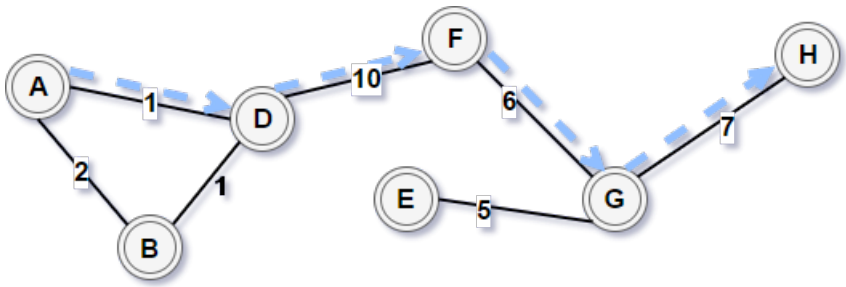

FIGURE 2.2 – Représentation d'un graphe à 7 sommets et 7 arêtes avec un chemin dessiné en bleu entre le sommet A et le sommet H : {A, D, F,G, H}.

Arbre :

Un arbre est un graphe non orienté dans lequel chaque nœud (sauf le nœud racine) n'a qu'un seul parent. On désigne comme **feuille** un nœud n'ayant aucun fils. Dans un arbre on définit la **hauteur** comme étant la longueur du chemin de la racine vers le nœud feuille le plus éloigné. On parle de **profondeur** quand il s'agit de la distance entre n'importe quel nœud feuille et le nœud racine. On parle d'**arbre uniforme** pour désigner un arbre fini de hauteur n dont tous les nœuds qui sont inférieurs en profondeur à n ont le même degré et où tous les nœuds de profondeur n sont des feuilles.

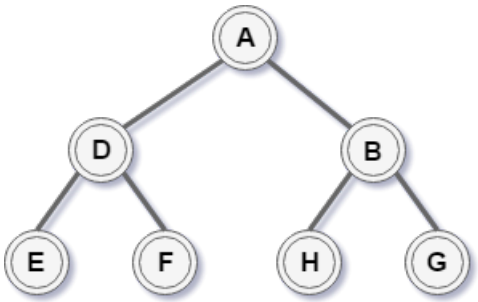

FIGURE 2.3 – Un arbre uniforme de profondeur 2 à 7 sommets et 6 arêtes

Le choix d'une représentation pour encadrer un problème se fait en fonction des contraintes et des données mais ce choix n'est pas unique et peut être différent en fonction de l'approche souhaité.

2.2.2 ET-OU graphe

Le graphe *ET-OU* (ou graphe de réduction de problème) est une modélisation destinée à représenter un problème comme étant la conjonction de plusieurs sous-problèmes qui peuvent être résolus indépendamment. Cette représentation est principalement utilisée lorsqu'il s'agit de trouver une stratégie de recherche efficace, c'est par exemple le cas si l'on souhaite résoudre le problème de la pièce contrefaite :

Nous avons douze pièces de monnaie et parmi elles se trouve une pièce contrefaite, c'est à dire qui est soit plus légère ou plus lourde que les autres.

L'objectif est de déterminer une stratégie pour identifier en au plus trois pesées (avec une balance) quelle est la pièce contrefaite. Il s'agit donc de sélectionner une suite d'actions de ce qui doit être pesé en premier pour avoir une chance d'identifier la pièce contrefaite. Bien entendu, ce problème peut être résolu en énumérant la totalité des solutions possibles mais l'objectif ici est d'utiliser une heuristique pour identifier une stratégie qui permet de résoudre ce problème en un minimum de pesée (action).

Pour résoudre ce problème, il faut décider du nombre de pièces à comparer à chaque pesée. On peut par exemple décider de peser les pièces une à une, deux à deux, trois à trois, et ainsi de suite. Intuitivement, on sait que si à la première action on compare un sous-ensemble de pièces restreint en prenant seulement deux pièces, l'approche sera plus hasardeuse puisqu'à la prochaine action, le sous-ensemble restant risque d'être trop important pour identifier une pièce contrefaite. Par contre, si nous pesons deux pièces au hasard en premier, il est possible d'obtenir le résultât en une seule pesée si l'une des deux est soit plus légère ou plus lourde.

Dans ce problème, nous ne sommes pas que confronté au choix de la prochaine action à entreprendre mais aussi aux conséquences de celle-ci qui affecteront inévitablement les prochaines décisions et délimiterons le prochain sous-ensemble.

Pour cela nous utilisons donc le graphe de réduction de problème où les nœuds représentent les sous problèmes et les arcs les conséquences de l'action entreprise sur ce sous-problème. L'avantage de ce type de modélisation est qu'il permet de découper le problème initial en sous-problèmes indépendants grâce à une technique appelée « diviser pour régner ».

Arêtes ET :

Mène à des sous-problèmes indépendants qui devront tous être résolus pour résoudre le problème associé au nœud père. Cet arc représente les changements dans la situations du problème.

Arêtes OU :

Mène à des sous-problèmes alternatifs, dont l'un devra être résolu pour résoudre le problème associé au nœud père. Cet arc représente les différentes réactions possible après un tel changement.

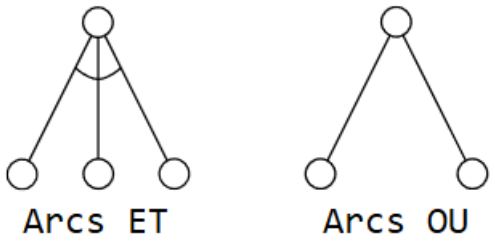

FIGURE 2.4 – Respectivement les arcs ET et OU du graphe de réduction de problème.

Nous pouvons donc modéliser notre problème sous forme de graphe où chaque décision prise depuis le nœud racine forme une solution possible résultant de la première action entreprise. Une solution n'est donc pas qu'un chemin du graphe, mais un sous-graphe de notre modèle commençant au nœud racine. La figure 1.5 donne un exemple d'une solution possible où l'action de comparer deux pièces de monnaies est prise en premier.

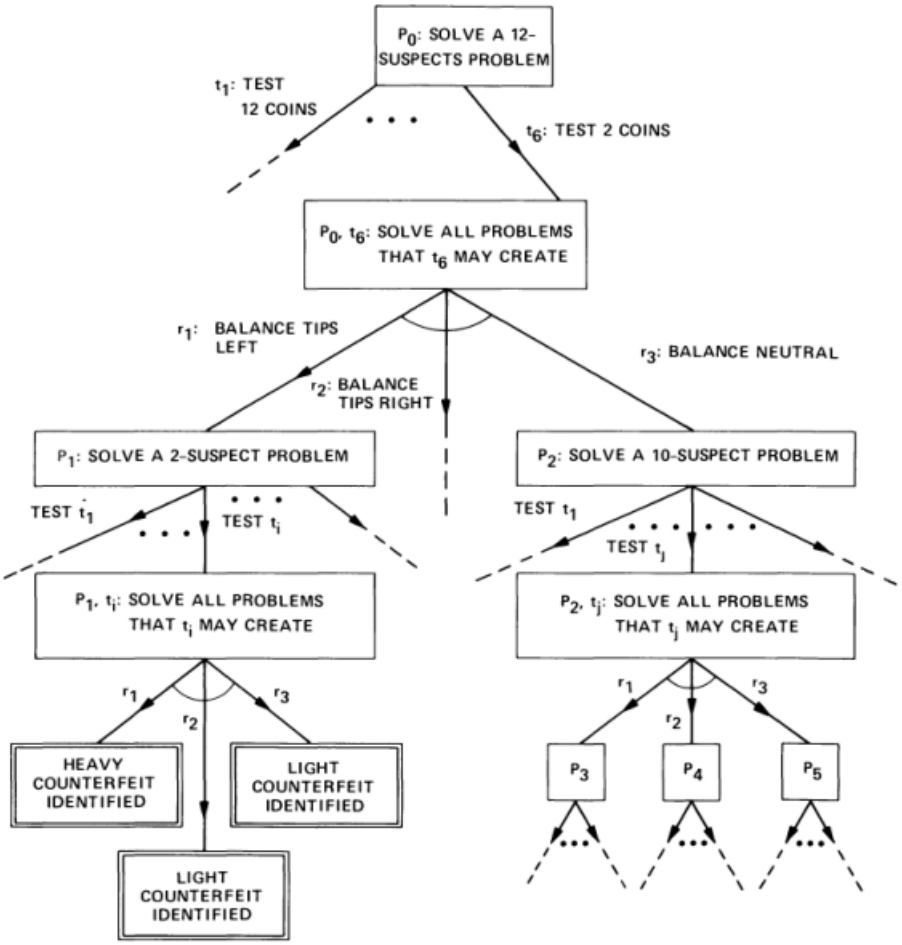

FIGURE 2.5 – Problème de la pièce contrefaite représenté avec un ET-OU graphe[11]

2.2.3 Représentation d'état

Une représentation d'état consiste essentiellement en un ensemble de nœuds représentant chacun les états possibles du problème. Les arêtes entre les nœuds représentent les actions possibles d'un état à un autre. Chaque représentation d'état prend la forme d'un graphe ou d'un arbre.

Une représentation sous forme d'espace-état sera plutôt utilisé pour modéliser un problème de satisfaction de contraintes ou de recherche de chemin. Si la solution peut être exprimée comme une séquence d'actions inconditionnelles ou comme un seul objet avec un ensemble de caractéristiques nous avons un problème de plus court chemin ou de satisfaction de contraintes qui est donc modélisable montrer ci-dessous.

Avant de représenter notre problème, il faut préalablement définir un ensemble de facteurs :

— Quel est l'objectif à atteindre ?
— Quelles sont les actions possibles ?
— Quelles informations doivent être représentées dans la description des états ?

Par exemple on peut souhaiter rechercher des erreurs dans un programme. Pour cela, nous pouvons modéliser chaque nœud comme étant un état du programme issu des différentes conditions de branchement où les valeurs concrètes en mémoire seraient remplacées par des valeurs symboliques. Il s'agit alors de parcourir le graphe, pour identifier d'éventuelles valeurs pour lesquelles le programme n'aurait pas le comportement souhaité.

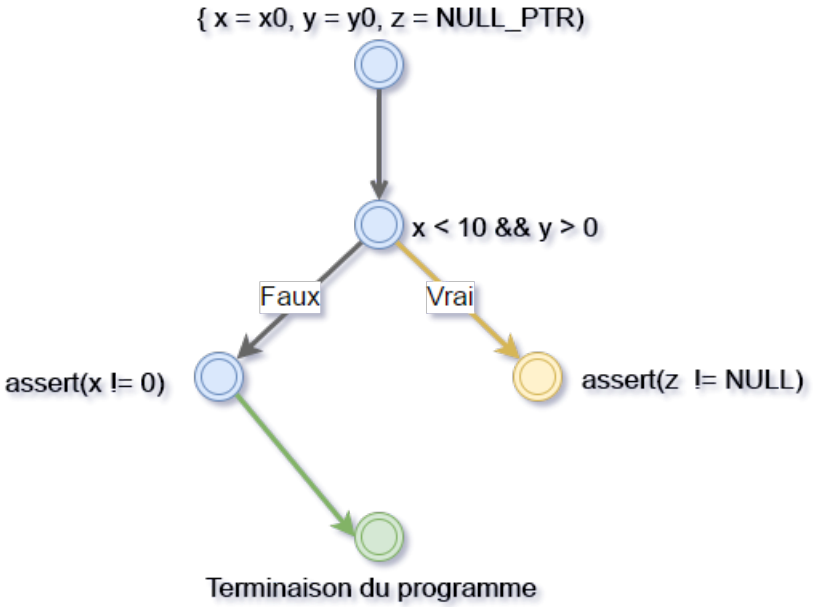

FIGURE 2.6 – Représentation sous forme d'état de l'exécution symbolique d'un programme

Chapitre 3

Procédures de recherches

Nous avons définie la notion d'heuristique de recherche, présenté quelques problèmes types rencontrés en y appliquant des heuristiques simples et intuitives ainsi que des contraintes rencontrées lorsque les données à parcourir sont trop importantes et développé différentes méthodes pour modéliser nos problèmes. Ce chapitre présente quelques algorithmes et méthodes connus pour parcourir et rechercher des propriétés dans nos graphes (modèles). Ces algorithmes définissent des stratégies ou politiques de recherche déterminant l'ordre dans lequel les nœuds du graphe seront parcourus afin d'obtenir la solution souhaitée ou une solution proche dans le cas d'heuristique.

Dans un premier temps nous présentons les procédures simples avec les algorithmes de parcours de graphe les plus populaires puis nous continuerons sur les recherches avec apprentissage par renforcement.

3.1 Procédures de recherches simples

Nous présentons deux grands types de stratégies que sont les stratégies informées ou à l'aveugle et les stratégies guidées ou informées.

3.1.1 Recherche non informée

On dit que la recherche n'est pas informée quand dans un graphe, la location de l'objectif n'altère pas l'ordre dans lequel les nœuds seront parcourus. Ces stratégies sont souvent inefficaces et peu pratiques dans le cas de larges problèmes mais elles ont l'avantages d'être simples et peuvent être appliquées à tout problème puisqu'elles ne prennent pas compte des données propres au domaine.

Parcours en profondeur (DFS)

Le parcours en profondeur donne la priorité aux nœuds les plus profonds du graphe et s'applique plus particulièrement au graphe de type arbre ou la profondeur est clairement définie. Pour les autres types de graphe, il faudra définir la notion de profondeur et la direction dans laquelle se diriger.

Ce type de parcours assure de trouver la solution, puisqu'en pire cas toutes les solutions seront énumérées ce qui rend cet algorithme peu efficace dans de nombreux cas.

backtracking :

Le *backtracking* (ou retour sur trace) est une variante du parcours en profondeur qui permet de revenir en arrière à chaque action afin de minimiser le nombre de sommets du graphe à parcourir quand on se retrouve dans certaines situations qui ne correspondent pas à l'objectif recherché (propriétés d'un nœud non conformes).

A chaque nœud, seulement un successeur est parcouru sauf si il ne remplit pas un critère donné. Dans le cas où il ne serait pas parcouru on revient à l'ancêtre le plus proche parcouru possédant au moins 1 sommet fils qui n'a pas été encore parcouru.

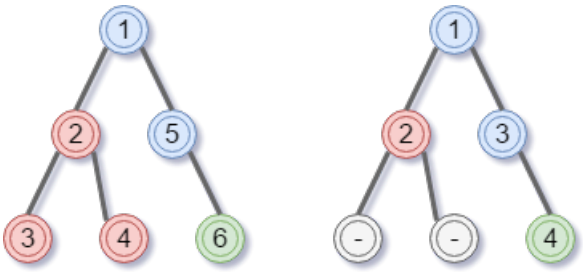

FIGURE 3.1 – Ordre dans lequel les sommets sont visités avec un parcours en profondeur (gauche) et sa variante (droite) : le backtracking.

Les problèmes d'optimisation ou de semi-optimisation tire parti de ce type de parcours. Si l'objectif est de trouver le coût minimal alors cette stratégie est efficace puisqu'elle permet de parcourir le graphe tout en maintenant le coût à tout instant t et ceci en priorisant les sommets les plus proches de la racine.

Parcours en largeur (BFS)

Contrairement au parcours en profondeur, le parcours en largeur donne la priorité au nœuds des premiers niveaux du graphe, c'est à dire aux sommets les plus proches de la racine qui seront parcourus en premier. Elle s'applique plus facilement aux arbres puisque la notion de niveau est naturellement défini, contrairement aux autres types de graphe ou il faudra définir ce qu'on entend par largeur pour orienter le sens du parcours.

Cette stratégie garantie aussi de trouver la meilleure solution possible puisque en pire cas, tout le graphe sera parcouru. C'est donc une méthode qui peut rapidement devenir très couteuse.

Procédure du coût uniforme :

La procédure du coût une uniforme est une variante du parcours en largeur. Au lieu de procéder au parcours des nœuds d'une même profondeur,

le parcours se fait en fonction du coût des nœuds. Chaque nœud est exprimé comme étant le coût du chemin menant à lui depuis le nœud racine et la stratégie est accomplie en parcourant toujours le nœud avec le coût le plus faible.

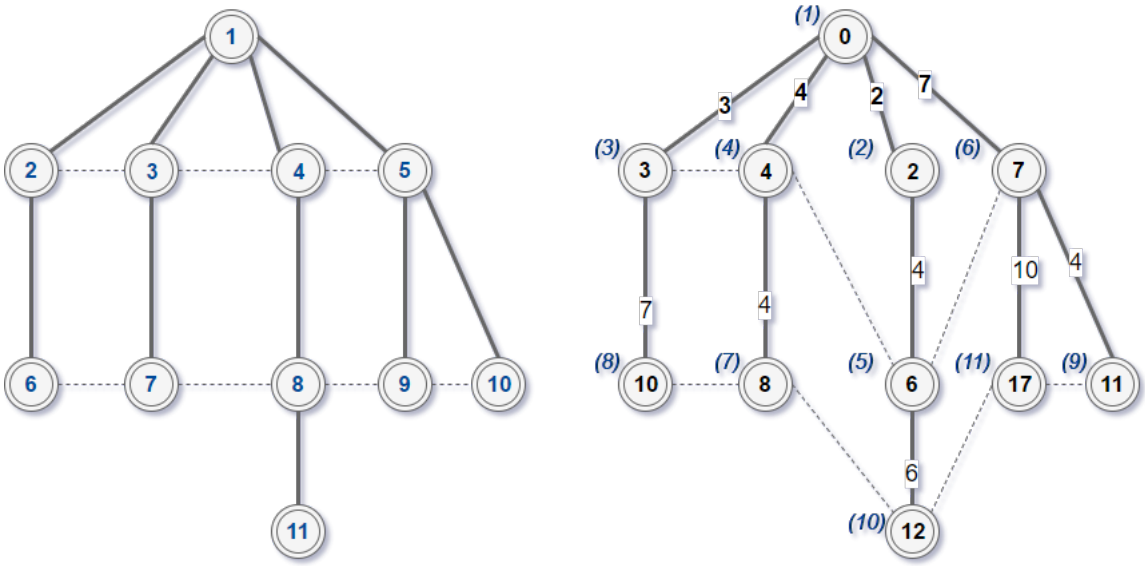

FIGURE 3.2 – Ordre dans lequel les sommets sont visités avec un parcours en largeur (gauche) et sa variante (droite) : la procédure du coût uniforme.

3.1.2 Recherche informée

Nous avons vu précédemment des stratégies qui ne prenaient pas en compte les données du problème autre que celles définies dans les sommets et les arêtes. Parfois il est possible de tirer parti d'informations qui ne sont pas dans le graphe pour diriger les recherches afin de parcourir les sommets qui ont l'air d'être les plus prometteurs. Cette section présente l'intérêt de combiner des méthodes (fonctions) heuristiques aux parcours classiques.

Best-First search (le meilleur d'abord)

Best-first search est un algorithme qui parcourt le graphe en étendant le nœud le plus prometteur d'abord. Contrairement aux heuristiques gloutonnes qui elles aussi sélectionnent les meilleurs éléments d'abord, *best-first search* le fait par évaluation, c'est à dire en estimant la valeur d'un nœud avant de le parcourir à partir de données du problème qui ne sont pas présent dans le graphe, comme une fonction heuristique qui permet d'estimer la valeur d'un nœud (récompense). De plus l'objectif de cette méthode n'est pas seulement de sélectionner le sommet le plus prometteur parmi un ensemble de nœud à prochainement parcourir mais de le sélectionner en comparant tous les sommets déjà rencontrés.

La promesse, c'est à dire l'estimation de la qualité d'un nœud est estimé grâce à une fonction heuristique $f(n)$ qui peut dépendre des données du

sommet n, de la description de l'objectif recherché, des informations récoltées lors du parcours de recherche et de toutes informations supplémentaires sur le domaine.

L'algorithme best-first décrit par Judea pearl[11]

1. Mettre le nœud racine (initial) dans la liste $NOEUDSOUVERTS$.
2. Si la liste $NOEUDSOUVERTS$ est vide, arrêter le programme : pas de solution.
3. Déplacer le nœud le plus prometteur de $NOEUDSOUVERTS$ c'est à dire celui pour lequel la fonction heuristique f est minimum (ou maximum dans un problème de maximisation) dans la liste $NOEUDSFERMES$. On nomme ce nœud n.
4. Si l'un des successeurs de n est l'objectif alors retourné le chemin vers ce nœud.
5. Pour chacun des successeurs n' de n :
   (a) Calculer $f(n')$.
   (b) Si n' n'était ni dans la liste $NOEUDSOUVERTS$ ni $NOEUDSFERMES$, l'ajouter dans la liste $NOEUDSOUVERTS$. Attaché l'évaluation de l'heuristique au nœud ainsi qu'un lien vers le nœud n.
   (c) Sinon comparer le résultât de la fonction heuristique avec le résultât enregistré (qui a été précédemment attaché au nœud). Si l'ancienne évaluation est meilleure (inférieure ou supérieure selon le problème de minimisation ou maximisation) alors passer à l'étape suivante. Si l'ancienne évaluation est moins bonne, remplacer l'évaluation ainsi que le lien pour pointer vers le nœud n. Si le nœud n est dans la liste $NOEUDSFERMES$, le déplacer dans $NOEUDSOUVERTS$.
6. Retourner à l'étape 2.

Algorithmes best-first spécialisés

Pour explorer un minimum de sommets dans un graphe à la recherche d'un chemin optimal, un algorithme de recherche doit constamment faire les choix les plus informés possibles pour décider du prochain nœud à explorer.

L'algorithme *best-first search* n'est que la patron d'une stratégie à appliquer et nécessite d'être plus amplement définie pour être implémenté concrètement. Il ne spécifie pas comment la fonction heuristique f est calculé ni d'où les informations pour décider quel est le meilleur choix possible proviennent ou même comment elles se propagent sur le graphe pourtant se sont les éléments indispensables à une recherche efficace.

A* :

L'algorithme A^* est un algorithme de type *best first* et est aussi une extension de l'algorithme de Dijkstra[3], un des plus populaires pour résoudre le problème du plus court chemin.

L'algorithme A* définie[12] sa fonction d'évaluation comme suivant :

$$f(n) = g(n) + h(n)$$

où $g(n)$ est le coût d'un chemin optimal du sommet racine s vers le sommet n et où $h(n)$ est le coût d'un chemin optimal du sommet n vers le sommet a, un éventuel nœud objectif.

L'algorithme A*[12]

1. Mettre le nœud racine (initial) dans la liste $NOEUDSOUVERTS$.
2. Si la liste $NOEUDSOUVERTS$ est vide, arrêter le programme : pas de solution.
3. Déplacer le nœud le plus prometteur de $NOEUDSOUVERTS$ c'est à dire celui pour lequel la fonction heuristique f est minimum dans la liste $NOEUDSFERMES$. On nomme ce nœud n.
4. Si n est l'objectif alors retourné le chemin vers ce nœud.
5. Pour chacun des successeurs n' de n :
   (a) Si n' n'est pas dans $NOEUDSOUVERTS$ ou $NOEUDSFERMES$, estimé $h(n')$ (une estimation du coût du meilleur chemin de n' vers un éventuel nœud objectif) et calculer $f(n') = g(n') + h(n')$ où $g(n') = g(n) + c(n, n')$ and $g(s) = 0$ où s est le nœud racine.
   (b) Si n' est dans $NOEUDSOUVERTS$ ou $NOEUDSFERMES$.
   (c) Si n' la condition de l'étape précédente est vraie et que n' est dans la liste de $NOEUDSFERMES$ alors déplacer n' dans $NOEUDSOUVERTS$.
6. Retourner à l'étape 2.

3.2 Procédures de recherches avec apprentissage

L'apprentissage automatique est un champs d'étude de l'intelligence artificielle. Appliquée à certaines méthodes systématiques elle permet d'exécuter des tâches complexes en mémorisant des informations obtenues lors de chaque étape. Il est par exemple possible d'utiliser cette méthode pour classifier, c'est à dire qu'on étiquette chaque donnée rencontrée en fonction de ses propriétés à des fins de regroupement ou de génération de nouvelles connaissances pouvant faciliter le traitement par des méthodes automatiques.

3.2.1 Monte Carlo Tree Search (MCTS)

Définitions

Monte Carlo Tree Search (MCTS) est avant tout une méthode pour la résolution pratique de jeux, son application permet de décider des prochaines actions (optimales) à jouer à partir du résultât de simulation faites sur des parties fictives et des actions effectuées par l'adversaire. Le processus du MCTS est le suivant : un arbre est construit de manière incrémental et asynchrone et pour chaque itération de l'algorithme, une politique est utilisée pour déterminer quel est le prochain nœud à parcourir depuis l'état actuel. Cette

politique est appelée politique de l'arbre et son rôle est d'équilibrer l'exploration de nouveaux nœuds et l'exploitation des nœuds déjà parcourus pour maximiser les informations récoltées jusqu'alors. Puis une simulation est effectuée depuis le dernier nœud sélectionné pour évaluer sa distance avec l'objectif cherché (a t-on des chances d'atteindre l'objectif et en combien de coup pouvons nous l'atteindre ?). Ensuite, grâce au résultât de cette évaluation, tous les nœuds parcourus jusqu'à maintenant pour atteindre le nœud actuellement sélectionné sont mis à jour pour tenir compte du résultât de la simulation. Les nœuds parcourus contiennent donc des statistiques qui permettent d'évaluer leur potentiel. *MCTS* appartient à la famille des méthodes d'apprentissage par renforcement puisque les répétitions des itérations successives permettront à l'algorithme d'affiner ses connaissances du problème.

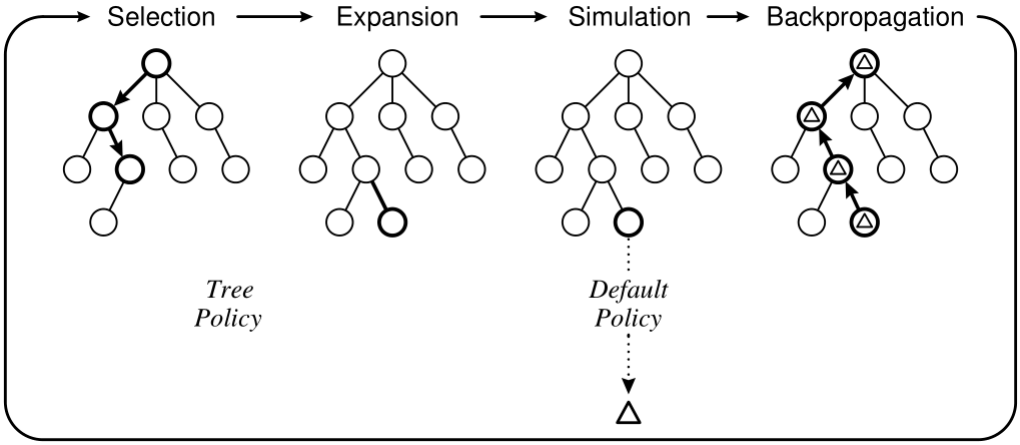

FIGURE 3.3 – Une itération du MCTS - source : *A Survey of Monte Carlo Tree Search Methods*[13]

Limitations

Les performances du *MCTS* dépendent donc du nombre de simulations produites puisqu'elles permettent à la recherche de s'orienter automatiquement vers les actions les plus prometteuses en fonction des données récoltées lors des simulations précédentes. Même si l'algorithme de base du *MCTS* s'est prouvé intéressant et efficace, le vrai bénéfice d'utiliser *MCTS* est atteint lorsque l'algorithme est adapté au problème donné. Les variantes du *MCTS* sont donc tout aussi intéressantes et il est important d'adapter le comportement général du *MCTS* face à chaque situation. De plus, pour effectuer les simulations il faut en mesure de simuler les résultats possibles depuis un nœud quelconque de l'arbre formant les actions possibles et leurs conséquences. Il faut donc être capable de connaître les actions qui peuvent être entreprises depuis le contexte actuel et c'est pour cette raison que cette méthode a eu un tel succès pour la résolution pratique de jeux.

Chapitre 4

Application

4.1 Contexte

4.1.1 Apprentissage de la programmation et tests

Maintenant que nous avons présenté différentes techniques de génération automatiques de tests, les tenants et aboutissants, la notion d'heuristique, les différentes méthodes de modélisation, des algorithmes de recherche simples et un plus complexes avec apprentissage par renforcement rappelons nos objectifs.

Dans le cadre de l'apprentissage de la programmation, nous souhaitons évaluer des programmes sources d'apprenants. Pour cela nous avons fait le choix de proposer une approche comparative, l'évaluateur soumet le code de la correction et les spécifications sous forme de formules mathématiques ou de simples phrases en langue naturel que doivent suivre les apprenants pour réaliser un programme qui exécute le comportement souhaité. Pour cela, nous mettons à disposition des outils qui facilitent la comparaison à l'exécution du programme soumis par l'apprenant au programme de la correction. Bien que ces tests soient très pratiques, leur écriture est encore laborieuse puisqu'elle demande un investissement conséquent et une certaine expertise. De plus, les tests sont souvent négligés par les évaluateurs et de nombreux programmes qui ne devraient pas être corrects le sont par manque de couverture des différents cas d'exécution. Nous souhaitons donc disposer d'outils pour générer des tests afin de détecter de manière automatisé les divergences de comportements entre programmes.

4.1.2 Limitations des méthodes existantes

Nous avons vu que de nombreuses méthodes existent et font l'objet de recherches intensives dans le domaine du génie logiciel dans un objectif d'améliorer la qualité des programmes testés en facilitant la génération et en améliorant la qualité des jeux de tests générés. Nous avons aussi remarquer que ces méthodes sont dans certains cas assez limitées et sont difficiles à réaliser dans des cas concrets.

C'est le cas de l'exécution symbolique qui est limitée face à l'explosion du nombre de chemins d'exécutions possibles d'un programme. Dans nos cas

d'utilisations c'est une contrainte que nous pourrions rencontrer fréquemment puisqu'une simple fonction avec des boucles imbriquées et des instructions conditionnelles suffit à mettre à bout bon nombre d'outils sur une machine standard. De plus, les contraintes de chemins dépendant d'opérations non linéaires (sinus, cosinus, multiplication, ...) contraignent aussi fortement la résolution des équations par les solveurs.

Des approches aléatoires sont aussi possibles mais sont souvent peu efficaces et plus le domaine d'entrée d'un programme est grand, moins les valeurs d'entrées générées auront une chance d'être pertinentes.

Les méthodes de *Search-Based Testing* proposent de contourner les contraintes rencontrées grâce à l'utilisation de procédures de recherches pour optimiser la génération des tests en fonction d'un objectif précis. Malheureusement, cette objectif ce résume bien trop souvent à un objectif de couverture de branche qui reste très limité puisqu'il ne tient pas compte du comportement du programme en fonction des valeurs d'entrées.

4.1.3 Objectifs

Comme nous disposons d'un cas très spécifique grâce à notre cadre d'apprentissage de la programmation, c'est à dire que nous avons une configuration avec plusieurs programmes ayant pour objectif de s'exécuter à l'identique du programme de la correction. Nous pensons qu'il serait judicieux d'utiliser une heuristique adaptée à notre cas en les complétant à des techniques de renforcement pour affiner et améliorer le parcours de chaque programme analysé.

4.2 Méthode proposée

4.2.1 Présentation

Nous possédons le code source du programme de la correction qu'on notera C et de n code source provenant de n apprenants ayant tentés de produire un programme suivant les consignes de l'exercice. Les programmes des apprenants sont note Pi où i représente le numéro du programme parmi l'ensemble.

Étape de filtrage

Nous définissons une première étape de test qui utilise k valeurs d'entrées générées aléatoirement mais suffisamment éloignées les unes des autres. Ces jeux de tests serviront à filtrer les programmes dont le comportement s'éloigne trop de ce qui était attendu. Au final, nous souhaitons garder pour l'analyse uniquement les programmes qui se rapproche de la solution, où il serait pertinent de trouver un ou plusieurs jeux de tests appropriés pour découvrir une divergence de comportement. L'approche aléatoire avec un jeu de données disparates et suffisant nous permet de nous conforter dans l'idée que ces programmes ne correspondent pas au résultat attendu.

Cette étape de filtrage est importante, puisque nous souhaitons minimiser la charge de calcul lors des prochaines analyses. De plus, comme nous souhaitons classifier les instructions des programmes des apprenants et les conditions de chemins il est important que ces instructions soient utiles à l'exécution pour effectuer le comportement souhaité. Au moins, en retirant les programmes qui sont apriori complètement erronés, nous retirons les potentiels programmes n'ayant aucun rapport avec l'exercice, ce qui permet de ne pas polluer la classification.

Toujours dans l'optique de filtrage, nous pourrions appliquer des techniques de *slicing* qui permettent de ne garder que le code nécessaire à une certaine utilisation mais ces techniques sont difficilement applicables à des programmes courts, ce qui représente la majorité des cas dans notre cadre. De plus, il faut noter qu'au préalable à l'analyse de comportement d'un programme, nous menons une analyse structurelle sur les programmes pour vérifier que certaines instructions sont bien présentes. Ces tests peuvent être éliminatoires et permettent de filtrer plus amplement les programmes.

Démarche d'analyse

Nous prenons un programme *Pi* au hasard parmi ceux ayant passé la première étape de filtrage et on applique notre analyse. Celle ci est un mixte entre l'exécution symbolique et l'heuristique du *Monte Carlo Tree Search* où nous utiliserons en fait l'exécution concolique lors de la simulation.

Notre premier programme est donc analysé symboliquement de la façon suivante :

Sélection

Nous allons faire des choix, dans un premier temps aléatoire, pour décider à chaque nœud (représentant la contrainte séparant des chemins d'exécutions) vers quelle direction nous allons. C'est donc la phase de sélection.

Génération

Une fois le nœud sélectionné, nous utiliserons une des bornes de la dernière contrainte rencontrée pour générer une première valeur d'entrée au programme satisfaisant à la limite une des bornes de la contrainte. Par exemple pour une contrainte où $x < 10$ et x un entier positif et paramètre d'entrée de la fonction, nous utiliserons $x = 9$. Si la contrainte exprimée par ce nœud, ne peut être affectée par une valeur d'entrée alors elle sera ignorée et nous passerons à la prochaine étape de sélection. Nous avons fait le choix de sélectionner des valeurs bornant les contraintes de chemins dans un premier temps car ce sont souvent sur les valeurs bornées que les apprenants se trompent lors de la rédaction de leurs programmes (exemple : $n - 1 < 10$ au lieu de $n < 10$).

Simulation

Grâce à cette valeur d'entrée que nous avons généré, nous pouvons l'injecter dans le programme de l'apprenant et de la correction pour détecter des éventuelles divergences de comportement, c'est une exécution concolique. On suppose d'ailleurs que si le programme plante pour une valeur donnée alors cela est considéré immédiatement comme une divergence. De plus c'est cette exécution concolique qui pourrait permettre de résoudre le problème de complexité des contraintes. Si elles deviennent trop importantes, elles pourraient être remplacées par des valeurs concrètes obtenus lors de l'exécution du programme et faciliter l'exécution des solveurs de contraintes.

Propagation

Une divergence dans notre cas correspond à un succès et une correspondance de comportement à un échec. Nous décidons donc si une divergence est détectée, de remonter l'information au nœud du chemin parcouru. Cette information, nous servira à classifier les instructions du langage menant à une divergence en supposant que ces instructions, pourront aussi mener à des divergences sur d'autres programmes. En parallèle de ça, nous mettons la valeur générée de côté pour en faire un éventuel jeu de test.

Une fois la simulation exécutée, nous retournons à l'étape de sélection et ainsi de suite jusqu'à ce que suffisamment de jeux de tests montrant une divergence soient trouvés ou que le temps de calcul pour l'analyse - qui devrait être fixé au préalable - est écoulé.

Une fois cette méthode posée, quelques questions se posent encore :
— si un nombre conséquent de jeux de tests montrant une divergence de comportement sur ce programme ont été trouvés, comment filtrer ceux qui le seront sur d'autres programmes ?
— comment transposer les données classifiées des nœuds des branches pour les appliquer sur d'autres arbres d'exécutions d'autres programmes ?

Sélection des valeurs générées

Dans le cas où de nombreuses valeurs seraient générées, nous pouvons simplement décider de n'utiliser que les valeurs de tests qui détectent une divergence sur au moins k programmes parmi n. L'avantage de notre méthode réside donc dans le nombre d'itérations de notre analyse sur tous les programmes pour y détecter des erreurs récurrentes. On peut imaginer que si tous les programmes erronés implémentent chacun un type d'erreur différent, alors notre méthode sera caduque.

Correspondance de la classification

Les contraintes de chemins d'un programme qui font l'objet d'une divergence auront été classifiés pour favoriser le parcours de ces chemins afin de

détecter d'autres erreurs dans d'autres programmes. Pourtant, il n'est pas dit que les autres programmes puissent correspondre en terme de contraintes puisque deux programmes peuvent implémenter la même fonctionnalité avec une structure interne complètement différente. Nous pensons tout de même qu'il existe quelques techniques pour valoriser les analyses précédentes. Nous pouvons dans un premier utiliser des techniques de renommage pour uniformiser les programmes. De plus, nous pouvons avec l'exécution symbolique associer un nœud avec le code associé et éventuellement y récupérer les instructions utilisées. Ces informations peuvent être précieuses puisqu'elles pourraient permettre d'identifier des *pattern* (patron) récurrents dans l'ordre d'exécution des instructions qui sont à l'origine d'erreur.

4.2.2 Problématiques restantes

Nous avons présenté une démarche tirant parti des techniques du domaine du test avec l'exécution symbolique et concolique associé à une heuristique de recherche avec apprentissage avec le *Monte Carlo Tree Search*. Cette présentation n'est qu'une ébauche, sa réalisation nécessite de maitriser des outils d'exécution symbolique qui sont souvent complexes et de pouvoir les instrumenter pour tenir compte de notre cas d'utilisation, ce qui demande un certain temps. Outre les problématiques associées à l'évaluation de la capacité de cette méthode à répondre à notre besoin, nous avons d'autres problématiques qui se posent aussi et qui devront être complétées.

La première est la détermination de l'ordre dans lequel les programmes sont analysés. Comme nous tirons parti des techniques de classification, il est important que les premiers programmes puissent nous apporter une quantité d'informations suffisantes pour guider les prochaines analyses. Nous pensons qu'il est important de limiter la quantité de programmes à analyser et c'est pourquoi nous avons présenté une technique de filtrage mais il doit aussi être possible de déterminer l'ordre dans lequel les programmes seront analysés selon certains critères. Nous pouvons par exemple, analyser le premier programme de manière aléatoire et si celui ci ne permet pas de détecter d'erreur, alors nous pourrions sélectionner le prochain programme selon sa distance avec le programme courant. Par distance on entend le nombre de différence qu'un programme aurait dans sa structure par rapport à un autre, comme nous pourrions exprimer la distancer entre deux chaînes de caractères. Ceci n'est bien sur qu'une brève proposition mais il peut exister des techniques plus efficaces.

Le nombre de simulation est déterminant pour s'assurer de la qualité du parcours du *MCTS*. Dans notre cas, nous ne simulons pas réellement puisque nous exécutons le programme avec des valeurs générés (conjointement avec le programme de la correction). Il est très probable que cette simulation soit problématique à cause du temps d'exécution possible des programmes. Bien qu'on soit dans un cadre d'apprentissage, il est possible que des programmes mal écrit soit particulièrement lent mais corrects. Il faut donc trouver des

solutions pour diminuer cette complexité, l'idéal serait bien sur de ne pas exécuter le programme mais de le simuler complètement.

Conclusion

Dans ce mémoire nous avons présenté les différentes méthodes de génération automatique de tests dont l'exécution symbolique qui nous semblait pouvoir correspondre à notre problème. Même si toutes ces techniques paraissent prometteuses, de nombreuses problématiques ont à chaque fois été rencontrées à cause de la complexité des programmes à analyser (explosion du nombre de chemin, complexité des contraintes, etc.). C'est dans l'optique de trouver des solutions qui nous permettent de générer des tests en un temps raisonnable en utilisant les connaissances récoltées par les analyses précédentes que nous avons décidé de parcourir l'état de l'art des heuristiques de recherche dans l'espoir de comprendre et de pouvoir appliquer une heuristique à une des méthodes de génération pour résoudre notre problème.

Les heuristiques d'apprentissage par renforcement qui se popularisent ces dernières années apparaissent prometteuses pour notre cas d'utilisation. Elles nous permettraient de tirer parti de la variété de nos programmes qui sont supposés exécuter tous un même comportement pour classifier les données et parcourir plus efficacement les arbres d'exécutions associés. C'est pour cette raison que nous nous sommes attardés sur le *Monte Carlo Tree Search* qui permettait de résoudre des problèmes complexes en un temps minimal tout en classifiant les données rencontrées et en proposant des parcours intelligents.

Dans la dernière partie, nous avons présenté une ébauche de méthode pour tirer parti de tous les éléments vu précédemment, c'est à dire de l'exécution symbolique et plus spécifiquement de l'exécution concolique et d'une heuristique inspirée du *Monte Carlo Tree Search*. L'application de cette méthode dans le but de l'évaluer reste problématique puisque les outils d'exécution symbolique sont complexes et qu'il est délicat de les instrumenter dans notre cas d'utilisation pour un initié. De plus, d'autres problématiques se sont posées pour tirer au maximum parti de la classification des données lors de nos parcours. Transposer les données d'un arbre d'exécution symbolique à un autre n'est pas aisé puisque des programmes peuvent appliquer des méthodes différentes pour résoudre un même problème. Il ne fait aucun doute que le travail à réaliser reste conséquent.

Bibliographie

[1] David Silver, Julian Schrittwieser, Karen Simonyan et AL. « Mastering the Game of Go without Human Knowledge ». In : *Nature* (2017). URL : https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge/.

[2] Hristina Palikareva, Tomasz Kuchta, Cristian CADAR. « Shadow of a Doubt : Testing for Divergences between Software Versions ». In : *Software Engineering (ICSE) 2016 IEEE/ACM 38th International Conference on* (2016).

[3] E. W. DIJKSTRA. « A Note on Two Problems in Connexion with Graphs ». In : (1959).

[4] E. W. DIJKSTRA. « Jean-François Pradat-Peyre, Jacques Printz ». In : (2017).

[5] Mark Harman, Bryan FJONES. « Search-based software engineering ». In : *Journal of Systems and Software, Volume 103, 2015, p. 266* (2001).

[6] Kiran Lakhotia, Phil McMinn, Mark HARMAN. « Automated Test Data Generation for Coverage : Haven't We Solved This Problem Yet ? » In : *Academic and Industrial Conference - Practice and Research Techniques* (2009).

[7] James C. KING. « Symbolic Execution and Program Testing ». In : (1976).

[8] Prof. J. L. LIONS. « ARIANE 5 Flight 501 Failure Report by the Inquiry Board ». In : (1996).

[9] Phil MCMINN. « Search-Based Software Testing : Past, Present and Future ». In : *2011 Fourth International Conference on Software Testing, Verification and Validation Workshops* (2011).

[10] Saswat Anand, Edmund K.Burke, Tsong Yueh Chen, John Clark, Myra B.Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil MCMINN. « An Orchestrated Survey on Automated Software Test Case Generation ». In : *The journal of Systems and Software* (2013).

[11] Judea PEARL. « Heuristics : Intelligent Search Strategies for Computer Problem Solving ». In : *Library of Congress Cataloging in Publication Data* (1984).

[12] Peter E. Hart, Nils J. Nillson, Bertram RAPHAEL. « A Formal Basis for the Heuristic Determination of Minimum Cost Paths ». In : *IEEE Transactions of systems science and cybernetics* (1968).

[13] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon SAMOTHRAKIS et Simon COLTON. « A Survey of Monte Carlo Tree Search Methods ». In : *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES* (2012). URL : http://mcts.ai/pubs/mcts-survey-master.pdf.

[14] B. Hailpern, P. SANTHANAM. « Software debugging, testing, and verification ». In : (2002).

[15] Rui Yang, Zhenyu Chen, Zhiyi Zhang, Baowen XU. « EFSM-Based Test Case Generation : Sequence, Data, and Oracle ». In : *Software Engineering (ICSE) 2016 IEEE/ACM 38th International Conference on* (2014).