UNIVERSITÉ PARIS NANTERRE

MÉMOIRE DE FIN D'ÉTUDES

MIAGE

Stratégies de recherche et application au test logiciel

Auteur :
Valentin BOUQUET

Tuteur :
MCF. Emmanuel HYON

19 juin 2018

Résumé

«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.»[6]

La génération automatique de tests est un domaine déjà bien développé qui permet aujourd'hui grâce à différentes analyse 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 valeurs de tests pertinentes pour s'assurer de la qualité du programme.

Ces approches sont malheureusement limitées de par la complexité grandissante des programmes à tester qui confronte 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 repose sur des modèles d'arbres qui ne peuvent être parcourus en intégralité à cause de leur taille. Les algorithmes de parcours actuellement utilisés repose souvent sur des stratégies simples comme le parcours en profondeur qui est connu pour être particulièrement couteux.

Ces dernières années, grâce la maturation des recherches sur les différentes stratégies et heuristiques de parcours de graphe (et notamment celles avec apprentissage) et des applications récentes sur des problèmes concrets comme l'ordinateur *AlphaGo* 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.

Dans ce mémoire, nous présentons la notion d'heuristique, les différentes modélisations possibles, les algorithmes de parcours populaires, des heuristiques d'apprentissage et plus particulièrement les heuristiques Monte Carlo Tree Search (MCTS) ainsi que des applications au test logiciel. Enfin nous regardons si il serait possible d'appliquer des heuristiques de recherches (intelligentes) pour la génération de tests et plus particulièrement à des fins de faciliter l'apprentissage de la programmation.

Mots clés : heuristiques, exploration de chemins, test logiciel, génie logiciel, exécution symbolique, monte carlo tree search...

Rermerciements

Cette année est l'aboutissement de 5 années d'études universitaires et je tiens donc à saluer tous les enseignants et chercheurs que j'ai pu côtoyer durant toutes ces années ainsi que tous les étudiants des formations classiques et en apprentissage, de la licence au master de la *MIASH* et de la *MIAGE* 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 laisse place à un avenir plus prometteur.

Je remercie 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 particulièrement François Delbot et 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 qu'ils m'ont apportés 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

«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.»[6][8][10][13][7][11][1]

La génération automatique de tests est un domaine déjà bien développé qui permet aujourd'hui grâce à différentes analyse 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 valeurs de tests pertinentes pour s'assurer de la qualité du programme[12] .

Ces approches sont malheureusement limitées de par la complexité grandissante des programmes à tester qui confronte 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 repose sur des modèles d'arbres qui ne peuvent être parcourus en intégralité à cause de leur taille. Les algorithmes de parcours actuellement utilisés repose souvent sur des stratégies simples comme le parcours en profondeur qui est connu pour être particulièrement couteux.

Ces dernières années, grâce la maturation des recherches sur les différentes stratégies et heuristiques de parcours de graphe (et notamment celles avec apprentissage) et des applications récentes sur des problèmes concrets comme l'ordinateur *AlphaGo* 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.

Dans ce mémoire, nous présentons la notion d'heuristique, les différentes modélisations possibles, les algorithmes de parcours populaires, des heuristiques d'apprentissage et plus particulièrement les heuristiques Monte Carlo Tree Search (MCTS) ainsi que des applications au test logiciel. Enfin nous regardons si il serait possible d'appliquer des heuristiques de recherches (intelligentes) pour la génération de tests et plus particulièrement à des fins de faciliter l'apprentissage de la programmation[3].

Ces dernières années des méthodes d'intelligence artificielles ont été appliqués dans de nombreux domaines, c'est le cas par exemple de l'ordinateur X qui a battu le joueur Y au jeu de GO grâce à un algorithme reposant sur la famille des Monte Carlo Tree Search, méthode de recherche basé sur l'apprentissage[2].

Émergence de solutions pratiques[4] car la recherche est disponible (30ans de recherche d'IA) et que l'aire de big data ont permis l'émergence de cas d'applications concrets.

Intérêt : utiliser les méthodes de recherches arborescentes informées permettant de trouver des solutions à un problème donné à forte combinatoire et ceci en un temps raisonnable.

Le test et la vérification logiciel est un domaine très important pour X, Y et Z. La vérification de logiciel et la génération de tests est souvent limité à cause de la complexité des programmes et des ressources limités des ordinateurs actuels.

Chapitre 1

Heuristiques de recherche

1.1 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[8]. 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.

1.1.1 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.

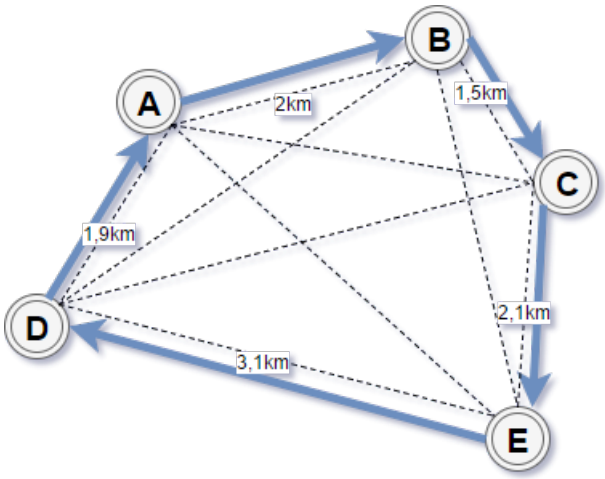

FIGURE 1.1 – Un chemin possible pour résoudre le problème du voyageur de commerce.

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 1.1 – 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).

1.1.2 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 aux 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
    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 meilleure 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 meilleure 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 1.2 – 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.

1.2 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 le représenter. 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[8] 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.

1.2.1 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.

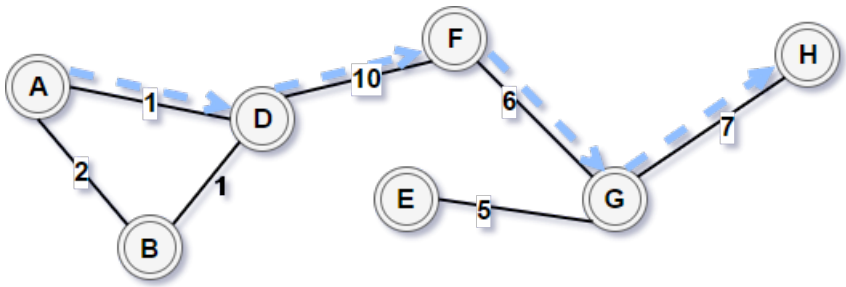

FIGURE 1.2 – Représentation d'un graphe à 7 sommets et 7 arêtes avec un chemin dessiné en bleu entre le sommet A et le sommet H : {A, D, F,G, H}.

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 inférieur en profondeur à n ont le même degré et où tous les nœuds de profondeur n sont des feuilles.

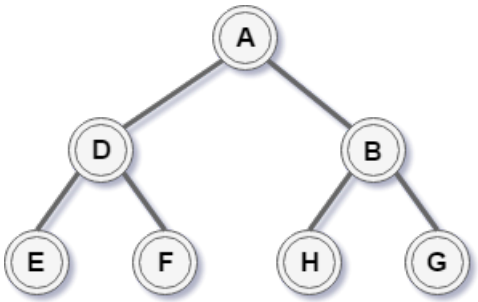

FIGURE 1.3 – Un arbre uniforme de profondeur 2 à 7 sommets et 6 arêtes

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.

1.2.2 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 permette 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 l'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.

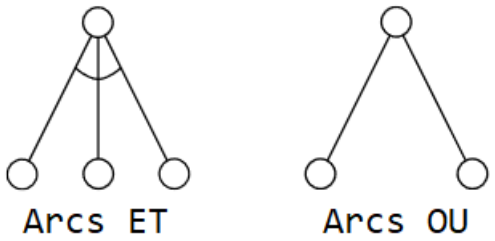

FIGURE 1.4 – Respectivement les arcs ET et OU du graphe de réduction de problème.

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 pas donc qu'un chemin dans le 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 2 pièces de monnaies est prise en premier.

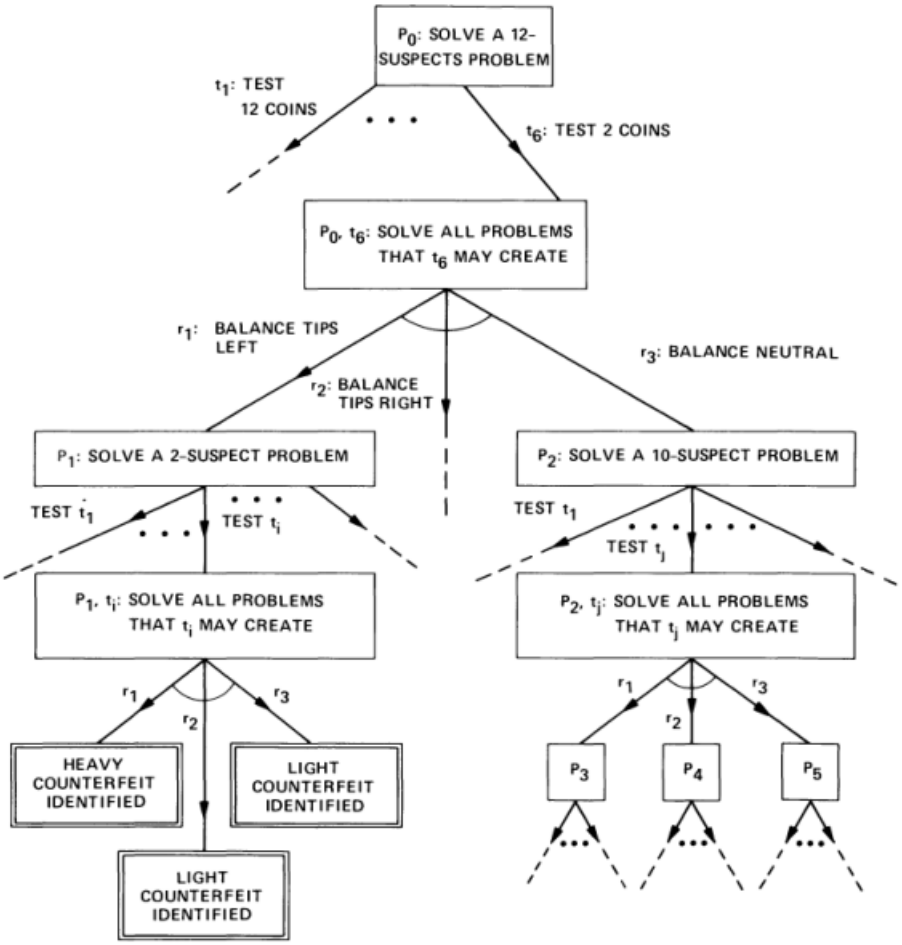

FIGURE 1.5 – Problème de la pièce contrefaite représenté avec un ET-OU graphe[8]

1.2.3 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'espaces états 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 comme ceci.

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

— Quel est l'objectif à atteindre ?
— Quels sont les actions possibles ?
— Quels 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.

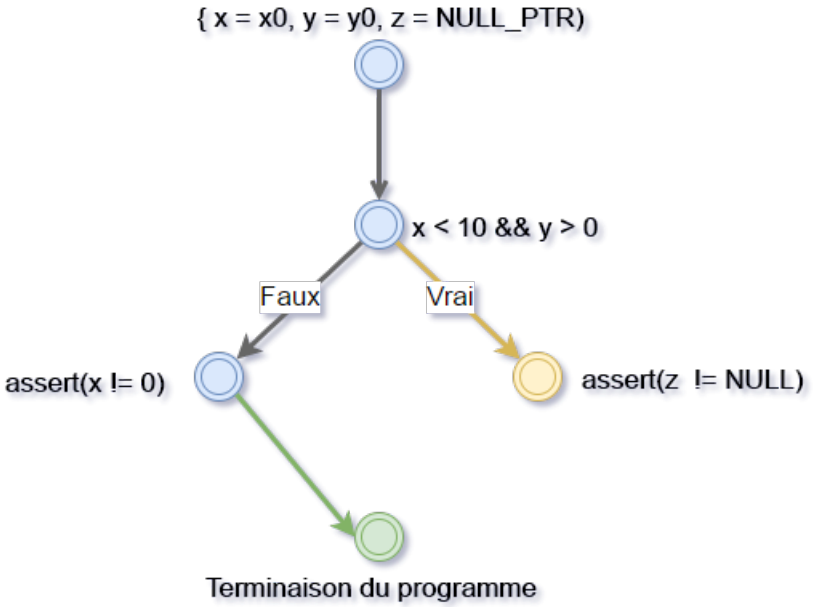

FIGURE 1.6 – Représentation sous forme d'état de l'exécution symbolique d'un programme

Chapitre 2

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 graphe (modèle). 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 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.

2.1 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.

2.1.1 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 conforme).

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.

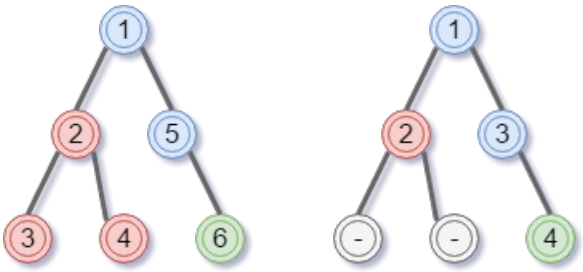

FIGURE 2.1 – Ordre dans lequel les sommets sont visités avec un parcours en profondeur (gauche) et sa variante (droite) : le backtracking.

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.

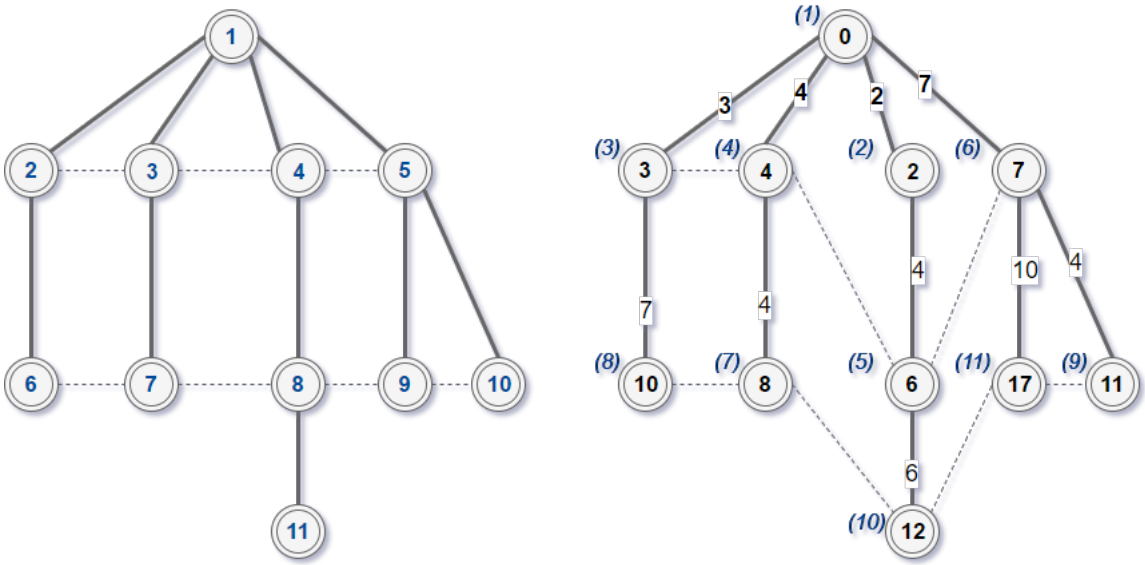

FIGURE 2.2 – Ordre dans lequel les sommets sont visités avec un parcours en largeur (gauche) et sa variante (droite) : la procédure du coût uniforme.

2.1.2 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[8]

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 possibles 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 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[5] un des plus populaires pour résoudre le problème du plus court chemin.

L'algorithme A* définie[9] 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*[9]

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.

2.2 Procédures de recherches avec apprentissage

2.2.1 Définitions

2.2.2 Monte Carlo Tree Search

Heuristique pour trouver des décisions optimales dans un arbre de décision.

Un arbre est construit de manière incrémental et asynchrone. Pour chaque itération de l'algorithme, une politique est utilisée pour trouver le noeud le plus important de l'arbre actuel. La politique d'arbre essaie d'équilibrer entre l'exploration de l'arbre (regarder dans les zones qui n'ont pas été essayées) et l'exploitation (regarder dans les zones qui ont l'air prometteuses).

Depuis un nœud une simulation est lancée et l'arbre se met à jour en fonction du résultât. Cela implique l'apparition d'un nœud correspondant à l'action entreprise par l'algorithme.

Les mouvements sont effectués pendant la simulation selon une politique d'arbre pré-défini par défaut qui dans le plus simple des cas consiste à effectuer des mouvements aléatoires uniforme.

MCTS n'a besoin que de l'état terminal de la simulation précédente pour effectuer la suivante. Il n'utilise pas les états intermédiaires. L'avantage est que cela réduit grandement les connaissances nécessaires à l'exécution de la méthode.

Même si l'algorithme est efficace sur une grande variété de problème, le vrai bénéfice d'utiliser MCTS est lorsqu'il est adapté au domaine du problème.

Méthode Monte-Carlo : Évaluation de la récompense :
$Q(s,a) = 1/N(s,a)$ ...

Définitions et intérêt

Famille d'algorithmes

Deux concepts fondamentaux :

- La valeur réelle d'une action peut être approchée en utilisant une simulation aléatoire.

- Ces valeurs peuvent être utilisées pour ajuster la politique vers une stratégie du meilleur d'abord (Best-First).

L'algorithme construit progressivement un arbre de décision guidé par les résultâts des explorations précédentes. L'arbre est utilisé pour estimer les valeurs associées à chaque mouvement.

Algorithme :

Basique : Construction itérative d'un arbre de recherche jusqu'à ce qu'un critère prédéfini soit atteint. Souvent une limite de de calcul comme le temps d'exécution, ou la saturation de la mémoire.

1. Sélection : depuis le noeud racine, une politique est appliquée pour sélectionner les noeuds pour atteindre le noeud le plus important à étendre (noeud non visité et non terminal).

2. Expansion : Un ou plusieurs noeuds son ajoutés pour étendre l'arbre (choix en fonction des actions disponibles).

3. Simulation : une simulation est appliquée depuis le nouveau noeud en fonction d'une politique par défaut pour produire un résultât.

4. Backpropagation : Le résultât de la simulation est remotée à travers les noeuds sélectionnés pour arriver au noeud ajouté pour mettre à jour ses statistiques.

Deux politiques :

- La politique de l'arbre : sélectionner ou créer un noeud feuille depuis les noeuds déjà parcourus/ajoutés.

- La politique par défaut : estimer la valeur d'un état non terminal (noeud ajouté) pour produire une estimation de sa valeur.

Applications

Chapitre 3

Applications au test logiciel

3.1 Le test logiciel

3.2 Une approche pour la génération automatique de tests

3.2.1 Exécution symbolique

3.3 État de l'art, tests et heuristiques de recherche

Conclusion

Bibliographie

[1] Alex Groce, Alan Fern, Martin Erwig, Jervis Pinto, Tim BAUER et Amin ALIPOUR. « Learning-Based Test Programming for Programmers ». In : *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2012). URL : https://link.springer.com/chapter/10.1007/978-3-642-34026-0_42.

[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] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin CHEN et Xuandong LI. « Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving ». In : *2016 31st IEEE/ACM International Conference on Automated Software Engineering* (2016). URL : https://ieeexplore.ieee.org/document/7582790/.

[4] L.J. White, E.I. COHEN. « A Domain Strategy for Computer Program Testing ». In : *IEEE Transactions on Software Engineering* (1980).

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

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

[7] Jinsuk LIM et Shin YOO. « Applying Monte Carlo Tree Search for Program Synthesis ». In : *International Symposium on Search Based Software Engineering* (2016). URL : https://link.springer.com/chapter/10.1007/978-3-319-47106-8_27.

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

[9] 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).

[10] 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.

[11] You Li, Zhendong Su, Linzhang WANG et Xuandong LI. « Steering Symbolic Execution to Less Traveled Paths ». In : (2013). URL : https://dl.acm.org/citation.cfm?id=2509553.

[12] 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).

[13] Chao chun Yeh, Han-lin Lu, Jia-jun YEH et Skih kun HUANG. « Path Exploration Base on Monte Carlo Tree Search for Symbolic Execution ». In : *Conference on Technologies and Applications of Artificial Intelligence (TAAI) (2017)* (). URL : https://www.computer.org/csdl/proceedings/taai/2017/4203/00/4203a033-abs.html.