UNIVERSITÉ PARIS NANTERRE

MÉMOIRE DE FIN D'ÉTUDES

MIAGE

Stratégies de recherche et application au test logiciel

Auteur :
Valentin BOUQUET

Tuteur :
MCF. Emmanuel HYON

18 juin 2018

Résumé

- Importances des tests pour la qualité logiciel
- Problématique : analyse et outil avec un complexité importante en temps d'exécution. Impossible de tout couvrir.
- Nécessite de contourner cette problématique en application des outils sur des logiciels en utilisant des propriétés qui font réduire l'ensemble des possibles.
- Possibilité d'appliquer des stratégies de recherches plus intelligentes pour détecter des erreurs ou des anomalies.

Mots clés : heuristiques, exploration de chemin, test logiciel, exécution symbolique...

Rermerciements

Merci, c'était super.

Table des matières

Introduction

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.

Émergence de solutions pratiques car la recherche est disponible (30ans de recherche d'IA) et que l'aire de big data ont permis l'émergeance 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.

A[8] B[9] C[12] D[5] E[10] F[1] G[7] H[3] I[6] J[2] K[4] L[11]

Chapitre 1

Stratégies de recherche

1.1 Heuristiques

1.1.1 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.

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.

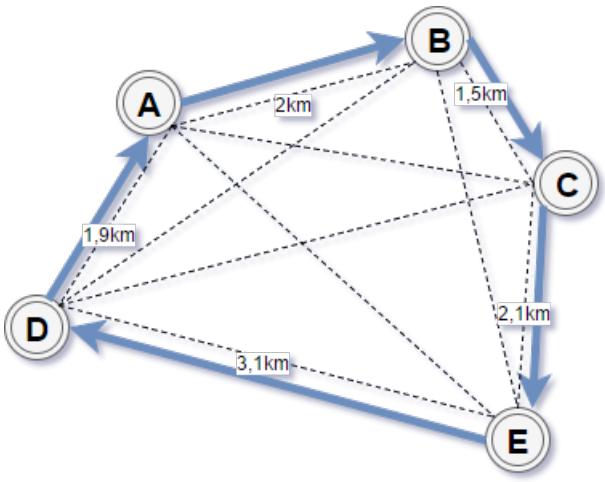

FIGURE 1.1 – Un chemin possible pour résoudre le problème du voyageur de commerce.

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 impacté 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	1 216 451 004 088 320 000

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).

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 $n-1$ distance à évaluer puis nous en aurons $n-2$ à la deuxième. Au final cet algorithme doit évaluer $\frac{n(n-1)}{2}$ distances.

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

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
```

Villes	Chemins	Chemins (algorithme glouton)
5	12	10
10	181400	45
15	43 589 145 600	105
20	1 216 451 004 088 320 000	190

TABLE 1.2 – Comparaison du nombre de chemins évalués pour la résolution du problème du voyageur de commerce avec un algorithme glouton (heuristique).

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.

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.

1.1.2 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 automatisé 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.

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.

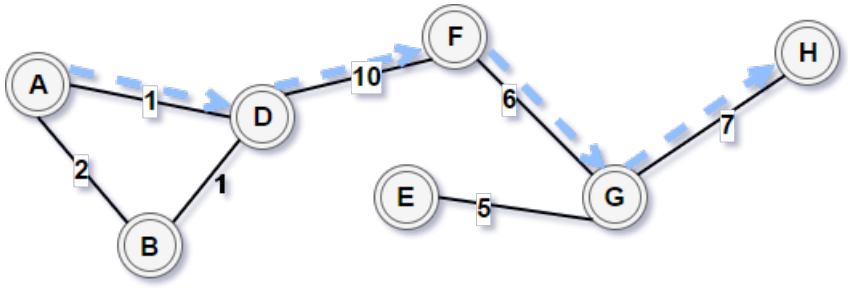

FIGURE 1.2 – Représentation d'un graphe à 7 sommets et 7 arêtes avec un chemin dessiné en bleu entre le sommet A et le sommet H A, D, F,G, H.

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.

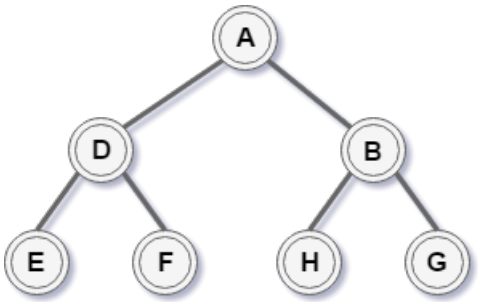

FIGURE 1.3 – Un arbre uniforme de profondeur 2 à 7 sommets et 6 arêtes

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.

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.

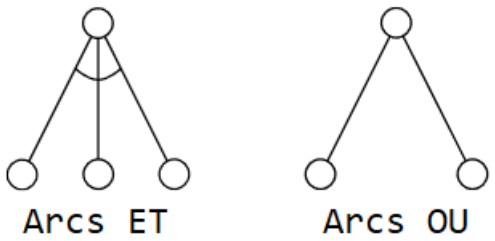

FIGURE 1.4 – Respectivement les arcs ET et OU du graphe de réduction de problème.

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.

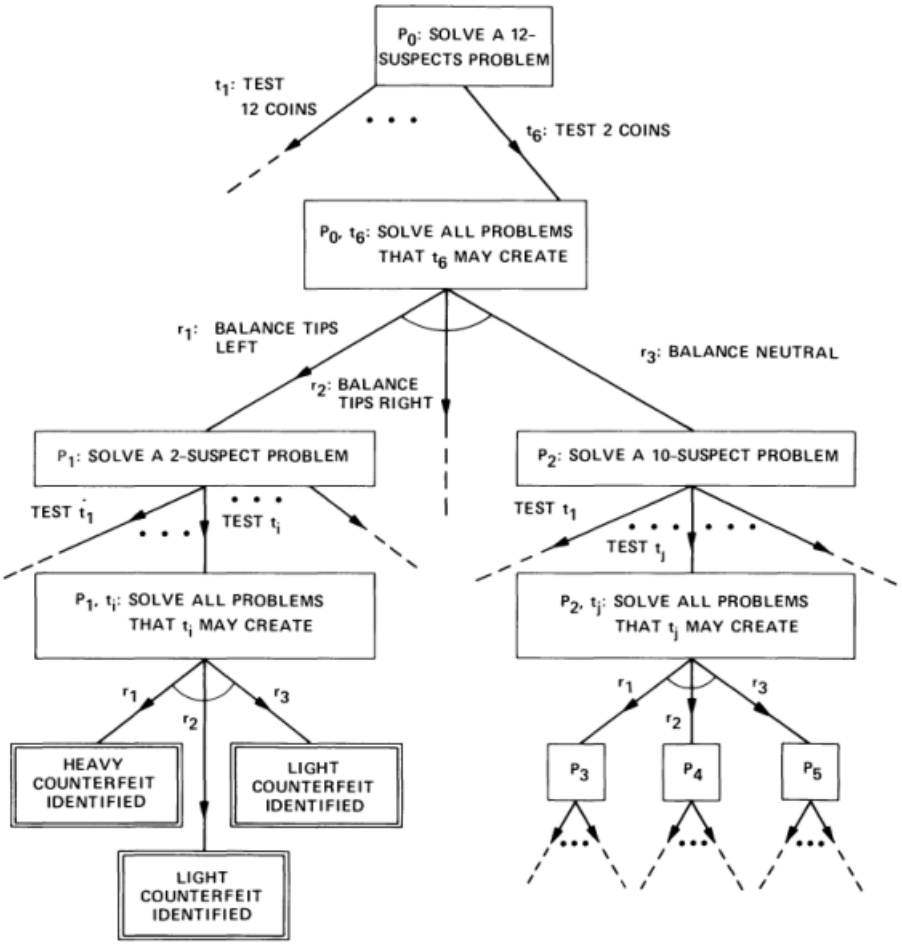

FIGURE 1.5 – Problème de la pièce contrefaite représenté avec un ET-OU graphe[8]

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.

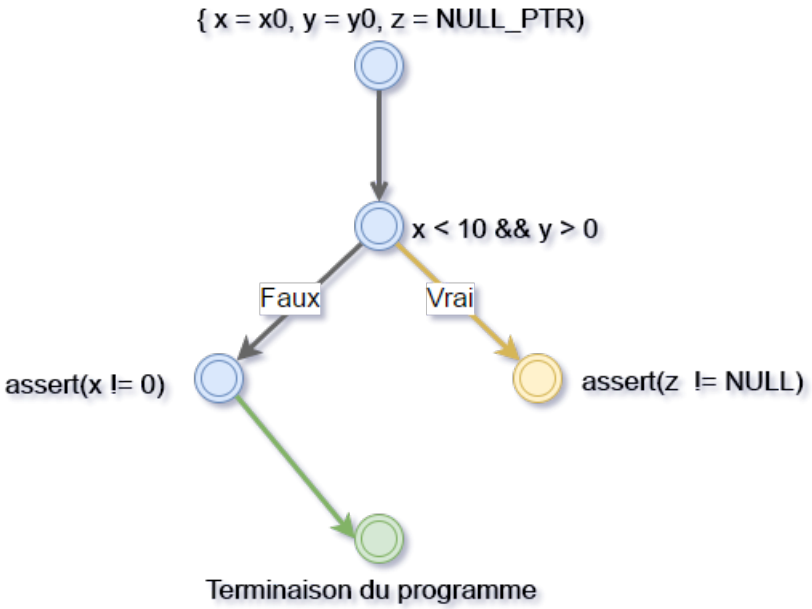

FIGURE 1.6 – Représentation sous forme d'état de l'exécution symbolique d'un programme

1.2 Procédures de recherches simples

Stratégie (politique) : procédure de recherche déterminant l'ordre dans lequel les nœuds du graphe seront parcourus afin d'obtenir la solution souhaitée (ou une solution proche). 2 grands types de stratégies que sont : stratégie à l'aveugle, stratégie guidé.

Recherche systématique non informé :

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 noeuds seront parcourus. Ces stratégies sont souvent inefficaces et peu pratique dans le cas de larges problèmes.

1.2.1 Recherche non informée

Parcours en largeur

Assigne une priorité plus importante pour les noeuds des premiers niveaux. Cette stratégie garantie de trouver une solution (la meilleure). Très couteuse par contre (on parcourt tout).

Variation : procédure du coût uniforme, variation du parcours en largeur. Elle ressemble beaucoup à l'algorithme de Dijkstra. les noeuds sont définis

comme étant le coût total du chemin pour aller jusqu'à lui et il s'agit de parcourir à chaque fois les descendants du noeud avec la plus petite valeur (si problème minimisation).

Parcours en profondeur

La priorité est donné aux noeuds les plus profonds du graphe. Variations : Sous une à deux conditions définis on permet d'arrêter le parcours et de revenir à un autre noeud. Par exemple il est possible de reconnaître un noeud comme n'ayant jamais de fin ou comme une impasse ou n'ayant pas une propriété satisfaisante.

Backtracking

C'est une version du parcours en profondeur qui applique la politique de dernier-entrée-premier-sortie (LIFO) pour la génération des nœuds au lieu de l'expansion. A chaque nœud, seulement un successeur est généré et parcouru sauf si il ne remplit pas un critère donné. Dans le cas où il ne serait pas parcouru on revient à l'ancestre le plus proche parcouru avec au moins 1 nœud non généré.

Intérêt : Problème d'optmisation et de semi-optimisation. Si l'objectif est de trouver le coût minimal alors cette stratégie fonctionne comme nous pouvons parcourir le graphe tout en maintenant le coût minimal parcouru à tout instant t.

Hill climbing

La plus populaire. Depuis la position courante, sélectionner la montée la plus rapide. Possibilité de trouver un chemin ne finissant jamais et sans solution. Comme on ne peut pas passer 2 fois sur un noeud, il faut fermer un chemin si il n'est pas correct et on ne pourra plus le reprendre même pour optimiser son parcours. Il est donc possible de parcourir un chemin à partir de toutes les branches de profondeur 1 sans trouver de solution.

Avantages :

- Quand on connait/possède des informations pour éviter les mauvais parcours et atteindre plus rapidement l'objectif fixé.

- Histoire sur les expansions commutatives : l'expansion d'un noeud ne compremet pas l'expansion des autres noeuds ou même de leurs descendants. Si une stratégie irrévocable est prise (mauvais chemin) alors cela n'affectera pas les prochains parcours.

1.2.2 Recherche informée

Best-First

L'étape la plus évidente pour utiliser les informations de l'heuristique est de décider du prochain noeud à étendre d'abord (comme pour hill-climbing

mais plus sophistiqué). Il s'agit de prendre le meilleure noeud en le comparant à tous les noeuds déjà rencontrés.

Depuis le premier noeud on étend tous les successeurs et on prend le noeud n tel que f(n) soit le minimum (si min recherché). Puis on continue comme ça en comparant tous les noeuds ouverts non visités et en maintenant un état du chemin sur les successeurs.

A*

C'est un algorithme dont l'objectif est de trouver une des meilleures solutions possibles. C'est donc une heuristique qui ne permet pas de toujours forcémment trouver la meilleure solution.

Il agit très similairement à l'algorithme de Dijkstra si ce n'est qu'il s'arête lorsque l'objectif est atteint (indépendemment du coût pour l'atteindre).

1.3 Procédures de recherches avec apprentissage

1.3.1 Définitions

1.3.2 Monte Carlo Tree Search

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

Algorithme : 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 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 noeud une simulation est lancée et l'arbre se met à jour en fonction du résultât. Cela implique l'apparition d'un noeu noeud 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 (je suppose qu'on parle d'état d'autres chemins/noeud qui auraient pu être évalués). L'avantage est que cela réduit grandement les connaissances nécessaires à l'exécution de la méthode.

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

Méthode Monte-Carlo : (Différent de MCTS) Evaluation de la récompose : 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

Des applications pour des problèmes du plus court chemin (problème du voyageur). Il est efficace pour le problème du voyageur canadien où certain des chemins peuvent être bloqués avec une certaine probabilité (utilisation d'une variante UCT).

1.3.3 Théorie de la décision

Elle combine les théories probabilistiques avec des théories utilitaires (heuristiques) pour offrir une approche formelle pour la prise de décision dans l'incertain.

Processus de décision Markovien : Modélise de manière séquentielle des problèmes de décision dans un environnement entièrement observable. Les décisions sont modélisées comme un ensemble d'état, action dans lequel chacun des prochains états est évalués grâce à une distribution de probabilité

en fonction de l'état courant et de l'action entreprise. Une politique est une correspondance entre états et actions en spécifiant quel action doit être entreprise depuis chaque état. L'objectif et de déterminer la politique qui permette de maximiser la récompose. La fonction de transition évalue la probabilité depuis l'état s et l'action a de se retrouver dans l'état s' (les états sont incertains).

Processus de décision Markovien partiellement observable (POMDP) : Contrairement à l'approche MDP, l'oracle n'a qu'une information partielle de l'état courant.

1.3.4 Les autres procédures

Chapitre 2

Test et stratégies de recherche

2.1 Le test logiciel (c'est quoi ? pourquoi ?)

2.2 L'exécution symbolique ?

Méthode pour simuler l'exécution d'un programme. Elle collecte les contraintes des branches de décision et remplace les valeurs des données en mémoire par des valeurs symboliques à base de formules mathématiques.

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

2.3.1 Article 1

Path exploration based on Monte Carlo Tree Search for symbolic execution

Veut répondre à la problématique de l'explosion des chemins lors du parcours du graphe d'exécution symbolique d'un programme. L'article compare les différents algorithme en fonction du nombre de blocs d'instruction parcouru.

2.3.2 Article 2

Monte Carlo Tree Search for program synthesis L'objectif de la synthèse de programme est de produire de manière automatique un exécutable d'un segment de code d'un programme qui garantis certains critères. Méthode la plus utilisé auparavent : genetic programming.

Chapitre 3

Application

3.1 Présentation de mon cas

3.2 Raisonnement

3.3 Application

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] 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.

[6] Saswat Anand, Edmund 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 : *Journal of Systems and Software (JSS)* (2013).

[7] Praveen Ranjan Srivastava, Amitkumar Patel, Kunal PATEL et Prateek VIJAYWARGIYA. « Test Data Generation Based on Test Path Discovery Using Intelligent Water Drop ». In : (2012). URL : https://dl.acm.org/citation.cfm?id=2603220.

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

[9] 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.

[10] 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.

[11] 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).

[12] 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.