Compte rendu SAÉ 2.02

Exploration algorithmique d’un problème

Igor GALLEA, Ilan MATHIEU, Yan Ait Ouazzou

1A2

Sommaire

Introduction

Dans le cadre du projet de résolution algorithmique d'un problème, nous avons décidé de choisir le problème du cavalier qui nous paraissait être le plus intéressant à traiter, consistant ainsi à trouver un chemin permettant à un cavalier d'échecs de parcourir toutes les cases d'un échiquier en ne passant qu'une seule fois par chaque case.

Nous avons ainsi cherché à aller plus loin que ce problème classique en cherchant à récupérer tous les chemins possibles à partir d'une case qui répond au problème du cavalier. Pour ce faire, nous avons exploré deux approches algorithmiques: tout d'abord, nous avons utilisé l'algorithme de backtracking pour trouver un seul chemin possible, puis nous avons adapté cette méthode pour découvrir tous les chemins envisageables. Ainsi ce compte-rendu expliquera nos débuts de piste de réflexion en utilisant la Programmation Orientée Objet pour ensuite utiliser une approche plus adaptée, puis nous verrons les différents algorithmes notamment avec l'utilisation de l'heuristique de Warnsdorff et enfin nous verrons l'utilisation de Pygame afin de produire une interface graphique pour la visualisation.

Approche en Programmation Orientée Objet (POO)

En premier lieu, nous avons essayé une approche en POO afin de pouvoir connaître facilement des paramètres propres à une case tels que: si elle avait déjà été visitée, si le cavalier était dessus, son numéro de case et ses voisins potentiels. Si cette approche nous paraissait intéressante au début, nous avons vite remarqué que cela nous compliquait grandement la tâche pour manipuler les informations nécessaires aux algorithmes et que le traitement de ces informations se faisait de façon bien plus naturel en utilisant une approche impérative au travers des listes.

Premier algorithme : Backtracking pour un seul chemin - Adaptation pour trouver tous les chemins

Si l'algorithme nous paraissait compliqué à imaginer au départ, nous nous sommes inspirés de l'algorithme de backtracking utilisé lors de la S1.02 où nous devions créer un algorithme de résolution de sudoku. Et afin de pouvoir l'utiliser dans ce problème, nous avons utilisé des concepts de la théorie des graphes tels que le parcours en profondeur. Ainsi, en mélangeant ces deux techniques, trouver un chemin où le cavalier passe par toutes les cases du plateau sans repasser deux fois sur la même fût relativement simple. Cela nous a tout de même donné quelques difficultés puisque fusionner des algorithmes différents n'est pas chose aisée et n'est certainement pas quelque chose dont nous avons l'habitude.

Si le travail était fait à 50%, il nous restait tout de même une autre partie du travail à faire, adapter notre algorithme pour qu'il ressorte tous les chemins possibles. Sur cette partie, ce fût non pas le travail de programmation le plus compliqué mais celui de réflexion. En effet, trouver un seul chemin était quelque chose mais tous les trouver était bien plus complexe et cela nous a pris du temps avant de trouver une solution. Ainsi, si le programme basique s'arrêtait au moment où il avait trouvé un chemin valide, ne trouverait-il pas tous les chemins s'il ne s'arrêtait pas ? C'est en partant de cette question que nous avons pu réussir cette étape. En effet, en adaptant le programme afin qu'il ajoute chaque chemin trouvé à une liste et en changeant certaines parties du code, l'algorithme nous

renvoyait une liste de tous les chemins possibles, mission réussie ! Seul problème, trouver un chemin prend du temps, alors tous les trouver est extrêmement long. Nous avons ainsi décidé d'améliorer la vitesse de notre programme de base en utilisant une heuristique.

Deuxième algorithme : Optimisation de l'exécution avec une heuristique

C'est en cherchant des façons d'améliorer la performance de notre programme de recherche d'un seul chemin que nous avons découvert le mathématicien H. C. Warnsdorff. (Pour information, nous n'avions que le premier algorithme qui renvoie un seul chemin à ce moment-là, le deuxième les renvoyant tous a été développé plus tard.) Nous nous sommes alors intéressés à ces travaux sur le problème du cavalier et avons découvert qu'il avait proposé une méthode au début du 19 siècle qui pouvait accélérer énormément la vitesse de recherche en se basant sur une simple règle: Le cavalier doit se déplacer vers la case voisine ayant le moins de possibilités de mouvement. Cette règle vise à réduire les chances de rencontrer des impasses et à maximiser les chances de trouver une solution. Ainsi, nous avons fait en sorte d'implémenter cette règle dans notre code. Pour se faire, nous trions les voisins en fonction du nombre de possibilités de mouvement disponibles puis en sélectionnant le voisin avec le moins de possibilités, cela nous permet ainsi de déterminer le prochain mouvement du cavalier.

Si cette heuristique paraît extrêmement simple, elle est néanmoins extrêmement efficace puisqu'elle nous a permis de passer d'une moyenne de 15s pour trouver un chemin à une solution instantanée. Si cela semble être extraordinaire, cette optimisation a des inconvénients que nous avons découvert en voulant l'implémenter dans l'algorithme pour trouver tous les chemins. En effet, celle-ci ne trouve pas tous les chemins. Cela paraît logique puisque son but est d'éviter les impasses et donc enlève l'approfondissement de certaines pistes qui pouvaient s'avérer correctes. De plus, c'est en calculant la durée du programme avec et sans l'heuristique que nous avons fait une découverte pour le moins surprenante. En prenant une case aléatoire d'un échiquier de 5x5, nous avions pris 70s à trouver tous les chemins possibles de façon classique. Seulement, en ajoutant l'heuristique, ce temps est passé à environ 200s soit près de 3 fois son temps classique.

Ainsi nous avons conclu que l'heuristique implémentée était extrêmement efficace pour la recherche d'un seul chemin mais qu'elle n'avait aucun intérêt pour la recherche de tous les chemins, rallongeant son temps et réduisant le nombre de chemin possible. Cela fut néanmoins très intéressant à étudier et très utile pour accélérer les phases de test pour le premier programme.

Interface graphique pour la visualisation

Afin de permettre aux utilisateurs lambdas d'avoir une meilleure compréhension du problème et une visualisation concrète de la solution de façon esthétique, nous avons décidé d'implémenter une interface graphique. Dans un premier temps, nous avons commencé par utiliser le module Tkinter, mais son utilisation étant très complexe pour peu de résultat, nous nous sommes dirigés vers un autre module bien plus intuitif d'utilisation: PyGame.

Nous connaissions ce module grâce à des projets passés mais n'étions néanmoins pas très à l'aise quant à son utilisation. Nous avons donc pu utiliser la grande documentation disponible en ligne pour réaliser la partie graphique. Nous avons fait le choix de pouvoir choisir la case du début non pas par la saisie de coordonnés mais par un clic à la souris, de sorte à ce que cela soit plus simple pour l'utilisateur.

Il n’était pas possible de montrer tous les chemins trouvés par l’algorithme alors nous avons fait le choix de faire s’afficher uniquement le premier chemin trouvé. Pour faciliter le suivi du parcours ainsi qu’une visualisation concrète à la fin, nous avons ajouté un tracé du chemin en vers et un assombrissement des cases par lesquels le cavalier était passé, rendant sa progression très visible. Enfin, nous avons décidé de rajouter une zone de texte affichant le nombre de chemin trouvé à partir de cette case.

Conclusion

En conclusion, ce projet de résolution algorithmique du problème du cavalier sur l'échiquier s'est révélé être une expérience extrêmement enrichissante. À travers nos différentes approches algorithmiques, nous avons non seulement exploré la complexité de ce problème classique, mais également découvert de nouveaux concepts et techniques de résolution.

L'approche en Programmation Orientée Objet (POO) nous a permis d'appréhender la modélisation des cases de l'échiquier de manière structurée, bien que nous ayons rapidement réalisé que cette méthode compliquait la manipulation des données nécessaires aux algorithmes. Le recours à des listes et une approche plus impérative s'est alors révélé plus adéquat.

L'utilisation de l'algorithme de backtracking pour trouver un seul chemin, puis son adaptation pour découvrir tous les chemins possibles, a été une étape cruciale de notre exploration. Fusionner des concepts de backtracking et de théorie des graphes n'était pas sans difficultés mais nous sommes parvenus à relever ce défi.

L'optimisation de l'exécution avec une heuristique, inspirée des travaux de H. C. Warnsdorff, a été une découverte majeure dans notre parcours. Bien qu'elle ait considérablement accéléré la recherche d'un seul chemin, son efficacité s'est avérée être un obstacle lors de la recherche de tous les chemins possibles. Cette observation nous a permis de mieux comprendre les compromis entre vitesse et exhaustivité dans la résolution de problèmes algorithmiques.

Enfin, l'implémentation d'une interface graphique avec Pygame a donné une dimension visuelle à notre projet, permettant une meilleure compréhension du problème et de la solution pour les utilisateurs. Ainsi, cette visualisation a été un ajout précieux pour illustrer le processus du cavalier sur l'échiquier.

En somme, ce projet nous a permis d'explorer différentes approches algorithmiques, de découvrir de nouveaux concepts, et de comprendre les nuances entre efficacité et exhaustivité dans la résolution de problèmes. Bien que cela fût difficile, chaque étape nous a permis d’apprendre énormément de choses aussi bien sur la programmation que sur la recherche et la réflexion pour trouver des solutions à un problème.