Dans cette Saé 2.02 de graphes, nous avons choisi de prendre le problème des 8 reines parmi les 3 problèmes proposés. Nous sommes tous les 2 amateurs d'échecs c’est pourquoi nous n’avons pas retenu le problème du postier chinois. Ensuite parmi les 2 restants, Pierre avait déjà entendu parler de ce problème en math lors de sa terminale. De plus, ce problème nous semblait plus intéressant car la dame est un pion qui offre toutes les possibilités de déplacement ce qui nous paraissait moins évident dans le sens (mais pas dans le fond).

Pour répondre au problème posé par le sujet de la Saé sur la comparaison d'algorithmes, nous nous sommes questionnés sur comment comparer 4 codes, de quelle manière et avec quels outils. Nous avons donc appris et codé une nouvelle méthode de recherche qui est le backtracking, ainsi que faire des recherches sur trois autres méthodes pour les comparer nôtre code. Il y a quatre philosophies entre les programmes, le premier créé par nos soins utilise le “backtracking”, le second code écrit par l’IA copilote sur le web “recherche locale”, les troisième trouver le web (sur le blog de **xavier nayrac**) utilise l’algorithme génétique puis enfin le dernier utilise le depth-first search trouver sur github (c’était un code open-source sans de réel source). Pour une lecture plus claire, nos 4 code on aussi été commenté par l’IA copilot (disponible gratuitement pour les étudiant). Voici le résultat de notre travail.

Le problème des 8 reines consiste à trouver une solution pour placer 8 reines dans un échiquier de 8x8 (taille réglementaire). Pour le résoudre, il faut placer une reines puis éliminer toute les cases ou celle-ci peut se déplacer. Ensuite il faut en placer une autre et répéter cette action jusqu’à ne plus pouvoir en placer.

**Nos recherches**

25/1: Découverte du sujet

26/1: Première recherche autour du sujet et des algorithme utilisé pour résoudre le problème

01/2: Création de notre algorithme utilisant le backtracking

07/2: Nous avons trouvé le code à comparer qui utilise la recherche locale

22/3: Nous avons pris la décision de rajouter un troisième code à comparer qui utilise “l’algorithme génétique”

03/4: Après les événements de l’IUT et donc du prolongement de la date de rendu, nous avons utilisé un 4e code utilisant “Depth-first search”

**Le backtracking**

Le backtracking est une technique algorithmique de résolution de problèmes qui utilise une recherche exhaustive en essayant toutes les solutions possibles pour un problème, en commençant par une solution partielle et en explorant chaque choix possible à chaque étape. Lorsque l'algorithme rencontre une impasse, c'est-à-dire qu'il n'y a plus de choix possibles, il revient en arrière et annule le choix précédent pour essayer une autre possibilité. Cette approche est répétée jusqu'à ce que toutes les solutions possibles aient été trouvées ou que la recherche soit arrêtée. Le backtracking est souvent utilisé pour résoudre des problèmes de recherche de solutions, tels que les problèmes de placement de pièces, les problèmes de coloriage de graphes et les problèmes de résolution de labyrinthes.

Avantages:

* Exhaustivité : Le backtracking permet de trouver toutes les solutions possibles d'un problème, à condition que la recherche soit menée jusqu'à son terme. Cela signifie que si une solution existe, elle sera trouvée par l'algorithme de backtracking.
* Efficacité : Le backtracking peut souvent trouver une solution plus rapidement que d'autres approches de recherche de solutions, en particulier lorsque le nombre de solutions potentielles est très élevé.
* Facilité d'implémentation : L'algorithme de backtracking est souvent facile à implémenter, car il s'appuie sur des concepts simples tels que la récursivité et la recherche en profondeur. Il est également facilement adaptable à de nombreux problèmes différents.
* Espace mémoire limité : Le backtracking utilise un espace mémoire limité, car il ne stocke que les informations nécessaires pour poursuivre la recherche de solutions, plutôt que de stocker toutes les solutions potentielles.
* Capacité à gérer les contraintes : Le backtracking est particulièrement utile pour résoudre les problèmes qui impliquent des contraintes, car il peut facilement revenir en arrière pour annuler des choix qui ne sont plus possibles en raison de ces contraintes. Cela permet d'éviter des calculs inutiles et d'améliorer l'efficacité de la recherche de solutions.

Désavantages:

* Complexité : La complexité de l'algorithme de backtracking peut être très élevée, en particulier pour les problèmes qui ont de nombreuses solutions potentielles. Dans certains cas, la recherche peut prendre beaucoup de temps et d'espace mémoire.
* Limitations de la mémoire : Bien que le backtracking utilise un espace mémoire limité, il peut parfois être nécessaire de stocker une grande quantité d'informations, en particulier pour les problèmes qui ont de nombreuses solutions potentielles. Cela peut poser des problèmes de limitation de la mémoire.
* Difficulté de choix de l'ordre des essais : Le choix de l'ordre dans lequel les choix sont explorés peut avoir un impact important sur la rapidité de la recherche de solutions. Cela peut être difficile à déterminer pour certains problèmes et peut conduire à des résultats suboptimaux.
* Impossibilité de trouver des solutions : Bien que le backtracking puisse être utilisé pour trouver toutes les solutions possibles, il peut arriver qu'il n'y ait pas de solutions, ou que la recherche de solutions prenne trop de temps.

**La recherche locale**

La recherche locale est une technique algorithmique de résolution de problèmes qui explore des solutions à partir d'un point de départ donné en effectuant des mouvements locaux, et en sélectionnant une solution voisine qui est meilleure que la solution courante. Le but est de trouver une solution optimale, ou une bonne solution, sans avoir à explorer toutes les solutions possibles. Les mouvements locaux peuvent être des opérations simples, telles que l'échange de deux éléments d'une solution, la suppression d'un élément ou l'ajout d'un nouvel élément. La recherche locale est souvent utilisée pour résoudre des problèmes d'optimisation combinatoire, tels que le problème du voyageur de commerce, le problème d'ordonnancement et le problème de sac à dos. La recherche locale est une méthode heuristique qui ne garantit pas la recherche de la solution optimale, mais elle peut être très efficace pour trouver des solutions acceptables dans un temps raisonnable.

**L' algorithme génétique**

L'algorithme génétique est une technique d'optimisation qui s'inspire de l'évolution biologique pour trouver des solutions à des problèmes complexes. Il consiste à générer une population de solutions aléatoires, puis à les faire évoluer à travers des opérations de sélection, de croisement et de mutation, pour obtenir des solutions de plus en plus adaptées au problème.

Dans ce code, nous représentons chaque solution comme une liste de N entiers, où chaque entier représente la position de la reine sur la ligne correspondante. Nous initialisons une population de solutions aléatoires, puis nous évaluons chaque solution en comptant le nombre de conflits. Nous sélectionnons ensuite les meilleures solutions pour le croisement, en utilisant une roulette de sélection pondérée par le score de chaque solution. Nous croisons ensuite deux solutions parents pour créer deux nouvelles solutions enfants, en utilisant une méthode de croisement en deux points. Nous mettons ensuite les solutions enfants en inversant aléatoirement deux positions. Nous répétons ce processus pendant un certain nombre d'itérations, en conservant les meilleures solutions de chaque génération.

**Depth-first search**

La technique appelée "depth-first search" (DFS) est une méthode de parcours d'arbre ou de graphe en profondeur. Elle consiste à explorer le plus loin possible dans l'arbre ou le graphe avant de revenir en arrière pour explorer d'autres branches.

L'algorithme de recherche en profondeur commence par explorer un nœud racine donné, puis visite récursivement tous les voisins de ce nœud. La recherche continue en explorant chaque nœud non visité jusqu'à ce qu'il n'y ait plus de nœuds à explorer dans cette branche particulière, puis la recherche revient en arrière pour explorer d'autres branches.

La recherche en profondeur est souvent utilisée pour résoudre des problèmes tels que la recherche de chemins dans un labyrinthe ou la résolution de problèmes de mots croisés. Elle est également utilisée dans des domaines tels que l'intelligence artificielle et l'apprentissage automatique pour la recherche de solutions optimales à des problèmes complexes.

Nous avons utilisé la backtracking car c’est une technique très répandue pour résoudre ce genre de problème, les avantages de cette technique nous ont aussi confirmé que cette technique était la plus adaptée.

Pour comparer 4 algorithmes il y a plusieurs facteurs à prendre en compte. La complexité de celui-ci a codé, et donc à modifier en cas d’évolution, en rapidité, en mémoire utilisé puis enfin la taille du code.

C’est sur ces points que nous nous sommes appuyés pour répondre à la problématique de cette Saé.

Ce tableau est donc selon nous le résultat de nos tests et nos évaluations, sur les problèmes des 8 reines.

| Nom du test | Complexité du code | | rapidité | | mémoire utilisé | | taille du code | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Code 1 | 4/10 | | 57 ms | | 2888 octets | | 1 867 octets | |
| Code 2 | 6/10 | | 63 ms | | 16528 octets | | 2 900 octets | |
| Code 3 | 8/10 | | 43 ms | | 448 octets | | 1 467 octets | |
| Code 4 | 10/10 | | 68 ms | | 19648 octets | | 1 915 octets | |
| Résultat test | Code 1 | | Code 3 | | Code 3 | | Code 3 | |

Pour évaluer la complexité du code, nous nous sommes basés sur des éléments simples d'algorithme: les outils utilisés et la période à laquelle elle nous ont été apprises lors de notre première année de BUT1 en informatique. Le résultat est sous la forme d’une note sur 10. Plus la note est élevée, plus le code est complexe à nos yeux dans la compréhension de celui-ci et donc de la possibilité de lui apporter des solutions.

Pour mesurer le temps d’exécution, nous avons utilisé un programme (timecompare.py), qui a mesuré le temps d'exécution en ms des programmes comparé, pour trouver une solution au problème.

Pour mesurer le temps d’exécution du code, nous avons utilisé un programme (memorycompare.py), qui a mesuré l’espace mémoire utilisé en kb des programmes comparés, pour trouver une solution au problème.

Pour mesurer la taille du code nous avons simplement regardé la taille du fichier. Selon, cet élément peut être à prendre en compte pour de très long code, ici cela peut paraître et est dérisoire mais nous avons quand même voulu l'évaluer pour avoir un schéma de comparaison utilisable pour tous les codes.

**Approfondissement**

Si nous allons plus loin dans notre raisonnement, en comparant les 4 algorithmes pas seulement sur le problème des 8 reines mais sur les problèmes des reines en général. On observe certains changements sur les “classement” dans lequel nos algo sont placés. Se baser sur 8 reines est intéressant du point de vue algorithmique car il permet de les comparer pour une valeur N fixe ainsi de les comparer sur l’aspect purement technique. Si on fait le lien avec le monde qui nous entoure, on cherche a faire en sorte que tout aille plus vite, toujours plus vite malgré l’augmentation de la complexité de chaque élément. Dans notre cas, nous avons regardé et classé les 4 algos sur les performances sur un plateau de 1000x1000. En utilisant un plateau si grand, des écarts qui sur un échiquier de taille réglementaire était très faible voir invisible sont mis à la lumière et on observe que l’algo le plus adapté selon nous a évolué. Dans notre conclusion, nous resterons rattaché à ce qui nous a été demandé mais par curiosité nous sommes allez voir si dans des cas plus poussé , nous aurions raisonné de la même manière.

**Conclusion**

Pour conclure, cette Saé nous a apporté plusieurs choses. Pour commencer, nous avons appris le fonctionnement du backtracking, comment l'utiliser et cela dans plusieurs langages et objectifs différents. Cette Saé nous a aussi permis de compléter notre apprentissage du python. Selon nous, plusieurs améliorations pourraient être apportées: une interface graphique plus complète et plus esthétique, l’utilisation d'autres possibilités de comparaisons. De plus, peut-être que “la recherche locale” n’est pas le meilleur ou le plus optimisé des algorithmes de recherche, ce qui signifie que notre comparaison n’est pas forcément optimale. Comparé notre algorithme de “Backtracking” à un algorithme plus efficace nous aurait donné une meilleure comparaison ainsi que plus de voies d’amélioration.