RAPPORT SAE 2.02 : EXPLORATION ALGORITHMIQUE

Nathan Thuault, Eliott Janot, Nidhish Ramane

1.1 - Le problème des huits reines

Ce problème consiste à placer huit dames sur un échiquier de 8x8 de manière à ce qu'aucune dame ne puisse en attaquer une autre. Une dame peut attaquer n'importe quelle autre pièce sur la même rangée, colonne ou diagonale. Le problème peut être généralisé à n dames sur un échiquier de n*n.

2 - Résolution

2.1 - Bruteforce

La première idée que nous avons eu pour résoudre le problème des 8 reines est plutôt simple. Nous générons et plaçons sur les échiquiers toutes les dispositions possibles de reines, puis nous analysons quels échiquiers obtenus valident le problème.

Pour ce faire, nous commençons par générer tous les arrangements possibles de 0 à n-1 (Ici n vaut 8). Chaque arrangement représente une disposition possible des dames sur l'échiquier. L'indice de chaque nombre dans l'arrangement représente la colonne de l'échiquier, et la valeur du nombre représente la rangée.

Pour réaliser cet arrangement, nous avons utilisé la fonction permutations() du module itertools. Voici un exemple pour n = 3 :

```
# permutations(range(3)) → 012 021 102 120 201 210
```

Dans cet exemple, le dernier échiquier est donc représenté par le dernier arrangement, 210. En établissant que les colonnes et les rangées commencent à 0, cela signifie qu'à la première colonne nous aurions une reine sur la troisième case de la rangée. En effet, le chiffre 2 se trouve à l'indice 0 (colonne) et sa valeur correspond à la dernière case 2 ou n-1 (rangée).

Ensuite, pour chaque arrangement, nous vérifions s'il représente une solution valide. C'est ce que fait la fonction possible(). Elle vérifie si deux dames sont sur la même rangée (ce qui signifie que les valeurs sont identiques) ou sur la même diagonale (ce qui signifie que la différence entre les indices est égale à la différence entre les valeurs).

C'est une approche simple mais pas très efficace, car elle doit générer et vérifier tous les arrangements, ce qui prend beaucoup de temps pour de grandes valeurs de n.

2.2 - Backtracking

Par la suite, une deuxième technique plus efficace semblait évidente. En effet, en appliquant un algorithme de backtracking, cela nous permettrait d'éviter de nombreuses solutions invalides.

Nous commençons par placer une reine sur la première colonne, dans la première case possible. Puis, nous passons à la colonne suivante et faisons de même. Si aucune case n'est possible, nous revenons en arrière (d'où le terme "backtracking") et déplaçons la reine de la colonne précédente à la case suivante possible. Cela nous évite de calculer toutes les possibilités qui découleraient de cette première erreur. Nous pouvons ensuite répéter ces étapes jusqu'à ce que toutes les dames soient placées. Cette technique garantit donc l'obtention d'une solution valide.

3 - Complexité algorithmique

Nous avons ensuite cherché à démontrer mathématiquement l'efficacité de la solution de backtracking face au bruteforce. Pour ce faire, nous avons calculé les complexités algorithmiques de ces deux méthodes.

3.1 - Bruteforce

Pour la génération des arrangements, le module itertools.permutations est utilisé. Il posséde une complexité de O(n!), où n est le nombre d'éléments à permuter.

Ensuite, pour chaque arrangement, la fonction possible() est appelée. La complexité de possible() est de O(n^2).

Ainsi, la complexité totale de l'algorithme de Bruteforce est O(n! * n^2), où n est le nombre de reines.

3.2 - Backtracking

La complexité algorithmique de ce programme dépend principalement de la fonction dames() car c'est la fonction récursive principale qui parcourt toutes les possibilités pour placer les reines sur l'échiquier.

La fonction possible() est appelée à chaque case de l'échiquier mais sa complexité est linéaire, ce qui est au maximum égal à la taille de l'échiquier `n`, donc sa contribution totale à la complexité est O(n).

La complexité de la fonction dames() dépend du nombre total de configurations qu'elle explore. Dans le pire des cas, elle explore toutes les combinaisons possibles d'emplacement des reines sur l'échiquier. Pour un échiquier de taille n*n, il y a n choix possibles pour la première colonne, puis pour chaque choix dans la première colonne, il y a au plus n-1 choix possibles pour la deuxième colonne, et ainsi de suite. Cela donne une complexité de l'ordre de O(n!) pour le pire des cas.

Cependant, grâce à l'optimisation fournie par la fonction possible(), le nombre réel de configurations explorées est bien inférieur à n!, cela réduit la complexité réelle. Mais dans le pire des cas, la complexité est O(n!).

4 - Conclusion

Pour conclure, le calcul des complexités algorithmiques de ces deux solutions nous permet de facilement les comparer. Lorsque nous choisissons une valeur de n assez petite, ces deux solutions sont relativement équivalentes. La différence de temps d'exécution ne semble donc pas significative pour 8 reines. Cependant, lorsque n tend vers +∞, O(n! * n^2) va croître bien plus rapidement que O(n!).