Christopher MAZZERBO L3 Mathématiques

Projet Taquin

Le but de ce projet est de s'intéresser à la solution du jeu du Taquin, connu aussi sous le nom de *15-puzzle* en anglais.
Celui-ci peut être représenté par un carré de $n \times n$ cases toutes numérotées de 0 à $n^2 - 1$, et on commence le jeu sur une configuration aléatoire. On considère que la case numérotée 0 est vide et qu'on peut l'échanger avec ses cases adjacentes à tout moment du jeu. Un échange avec une case sera considéré comme un *mouvement* et transformera l'état actuel en un de ses états *adjacents*. Naturellement, tout état a au plus 4 états adjacents.

Le but est d'obtenir la configuration où tous les nombres de 1 à $n^2 - 1$ sont numérotés dans l'ordre (on lit de gauche à droite puis de bas en haut) et où la dernière case est vide.
Avant de commencer, il est important de noter qu'un jeu du Taquin de taille n a n^2 cases, donc toute permutation de S_{n^2} est une configuration valide du jeu, d'où on en déduit le nombre de configurations possibles $Card(S_{n^2}) = n^2!$.
Ceci nous permet de réaliser qu'une configuration n'admet pas toujours de chemin à un autre car les signatures des permutations auxquelles elles correspondent doivent être la même.

Complexité - Dijkstra

Dans la première partie informatique de ce projet, on a créé un modèle naïf qui représente un graphe de tous les états possibles du jeu, et dont les arcs connectent tout état à ses états adjacents (longueur d'arc 1).
On utilise une implémentation d'un algorithme de Dijkstra (fournie dans un module *graph.py*)pour trouver une solution à partir d'un état initial, et on s'intéresse maintenant à sa complexité temporelle et spatiale.

Pour $n = 2$, la solution est trouvée quasi-instantanément pour tout état initial, tout état ayant au plus 2 états adjacents et le nombre total d'états pairs étant de $\frac{2^2!}{2} = 12$.

Pour $n = 3$, ce nombre s'élève à $\frac{3^2!}{2} = 181440$.
On remarque d'ailleurs que la solution est trouvée en une dizaine d'heures, ce qui est nettement plus long que pour $n = 2$, et semble cohérent lorsqu'on relève la formule $\frac{n^2!}{2}$ qui représente le nombre d'états pairs d'un jeu de taille n donné.
On détermine donc, avec une erreur d'estimation inévitable due au ralentissement de calcul d'un ordinateur qui doit gérer plus de données en même temps, une complexité temporelle de l'ordre d'$\mathcal{O}(n^2!)$.

La complexité spatiale est plus difficile à estimer car on ne peut pas directement mesurer l'espace qui a été utilisé pour résoudre un jeu de taille n.
Cependant, en considérant qu'on peut stocker un état sur quelques octets, pour $n = 2$, l'espace utilisé est presque négligeable (moins de 1Mo), et pour $n = 3$, on obtient une utilisation beaucoup plus forte des processeurs (visible dans un gestionnaire des tâches) et de la mémoire vive.

La complexité spatiale est, de la même façon que la complexité temporelle, de la forme d'une somme de $n^2!$ et d'un polynôme en n, donc par croissances comparées, on a bien sûr une complexité spatiale de l'ordre d'$\mathcal{O}(n^2!)$.

Questions préliminaires

On prend maintenant une approche différente à la résolution du jeu : on implémente un algorithme IDA*. Voici les réponses à quelques questions préliminaires présentes dans le sujet.

L'horizon représente une distance à l'état final. On peut voir cela dans la définition de l'horizon donnée :

$$h(c) = \sum_{\alpha=1}^{n^2-1} \left| c_\alpha^i - \left\lfloor \frac{\alpha-1}{n} \right\rfloor \right| + \left| c_\alpha^j - ((\alpha-1) \pmod n) \right|$$

Notons $c = (c_\alpha)_{\alpha \in [\![0, n^2-1]\!]}$, avec $c_\alpha = (c_\alpha^i, c_\alpha^j) \in [\![0, n^2-1]\!]^2$.
On peut démontrer rapidement que h', qui définit h par $h(c) = h'(c, c_f)$, est une métrique, où c_f est la configuration finale.
D'ailleurs, ici :

$$c_{f\alpha} = \left(\left\lfloor \frac{\alpha-1}{n} \right\rfloor, (\alpha-1) \pmod n \right)$$

On notera $c_k = (c_{k\alpha})$ pour $1 \leq k \leq 3$ trois différentes configurations.

— On a bien $h(c_1, c_2) \geq 0$:

$$h'(c_1, c_2) = \sum_{\alpha=1}^{n^2-1} \underbrace{\left| c_{1\alpha}^i - c_{2\alpha}^i \right|}_{\geq 0} + \underbrace{\left| c_{1\alpha}^j - c_{2\alpha}^j \right|}_{\geq 0} \geq 0$$

— On a bien $h(c_1, c_2) = 0 \Leftrightarrow c_1 = c_2$ par conséquence de l'équation écrite précédemment. Bien sûr $h(c_1, c_1) = 0$ est évident ici.

— Inégalité triangulaire :

$$\begin{aligned} h'(c_1, c_2) + h'(c_2, c_3) &= \sum_{\alpha=1}^{n^2-1} \left| c_{1\alpha}^i - c_{2\alpha}^i \right| + \left| c_{1\alpha}^j - c_{2\alpha}^j \right| + \sum_{\alpha=1}^{n^2-1} \left| c_{2\alpha}^i - c_{3\alpha}^i \right| + \left| c_{2\alpha}^j - c_{3\alpha}^j \right| \\ &= \sum_{\alpha=1}^{n^2-1} \underbrace{\left| c_{1\alpha}^i - c_{2\alpha}^i \right| + \left| c_{2\alpha}^i - c_{3\alpha}^i \right|}_{\geq \left| c_{1\alpha}^i - c_{3\alpha}^i \right|} + \underbrace{\left| c_{1\alpha}^j - c_{2\alpha}^j \right| + \left| c_{2\alpha}^j - c_{3\alpha}^j \right|}_{\geq \left| c_{1\alpha}^j - c_{3\alpha}^j \right|} \\ &\geq \sum_{\alpha=1}^{n^2-1} \left| c_{1\alpha}^i - c_{3\alpha}^i \right| + \left| c_{1\alpha}^j - c_{3\alpha}^j \right| = h'(c_1, c_3) \end{aligned}$$

La métrique h' mesure la distance d'une configuration à une autre, et permet ici de déterminer si une configuration est proche de la dite *solution* qui est une configuration c_f explicitée au-dessus (vérification triviale).

L'espace nécessaire pour IDA* est bien inférieur car tous les chemins explorés sont de taille inférieure au chemin le plus court qui donne une solution (explication dans la partie suivante).

Test

Pour $n = 2$ et un état $(3, 1, 0, 2)$:
`dijkstra()` trouve une solution en ~ 0.00099s.
`ida_star()` trouve une solution en ~ 0.00799s.
En faisant le test pour d'autres états, on remarque qu'`ida_star()` est environ 8 fois plus lent que `dijkstra()` pour $n = 2$.

Pour $n = 3$, c'est plus difficile de mesurer la différence entre les deux car `dijkstra()` prend beaucoup plus de temps, mais en faisant afficher combien de temps la fonction prend pour chaque tranche de 100 états, on peut estimer le temps total qui serait pris.
Pour un état donné qu'on peut représenter par $(0, 3, 5, 8, 6, 1, 2, 7, 4)$, pour 1000 sur les 9! états, on a une moyenne de temps d'environ 9s pour chaque tranche de 100 états « totalement explorés ».
Ceci nous donne :

$$\begin{aligned} 9! \times \frac{9}{100} &= 362880 \times 0.09 \\ &\approx 32660s \\ &\approx 9h \end{aligned}$$

On peut comparer cela à `ida_star()` pour $n = 3$ maintenant, qui trouve un résultat en ~ 0.5275s, et réaliser que cet algorithme est beaucoup plus optimal.

Voici un lien GitHub qui contient tout le code et les `gifs` générés pour animer la solution trouvée.

Le chemin obtenu avec `ida_star()` est bien optimal car l'algorithme cherche à partir de l'état initial, l'ensemble des états voisins (arcs, longueur de chemin 1). Ensuite, il cherchera pour chacun de ces voisins l'ensemble des arcs, ce qui donnera l'ensemble des chemins de longueur 2 à partir de l'état initial. On itère ensuite sur toutes les longueurs de chemins jusqu'à ce que tous les chemins possibles aient été parcourus, ou qu'on ait trouvé un chemin vers l'état final. Ce chemin sera nécessairement un chemin optimal, car l'algorithme a déjà parcouru l'ensemble de tous les chemins plus courts à partir de l'état initial.