Costa da Quinta, Joao Filipe

Bonus 5 algorithme

1.1)

1)

La méthode Branch-and-Bound (BaB) est une méthode qui consiste à l'exploration d'un arbre en explorant chaque E_node (branche), le E_node peut être choisit par l'ordre d'arrivée (FIFO/LIFO), ou alors, à l'aide d'une fonction Least Cost, cette fonction nous indique, intelligemment, quel E_node choisir, en associant à chaque E_node une approximation de coût, le but étant de suivre le E_node avec le plus petit coût affin de trouver le nœud réponse (nœud qui vérifie la propriété P) le plus court.

2)

Rôle h(x) est la profondeur de x, ça veut dire combien d'étapes on a déjà fait pour arriver à cet E_node, vu qu'on cherche un E_node réponse qui a le plus petit de cout possible, il est logique qu'on donne un « poids » à chaque pas qu'on a déjà fait.

Rôle g(x), on sait qu'un mouvement peut seulement remettre en place qu'une case, donc, si un board a 3 cases mal placés, alors on aura besoin de au moins 3 mouvements pour résoudre ce board.

h(x) compte le nombre de coups déjà fait, et g(x) est le minimum de coups qu'il nous reste à faire.

3)

Soit la liveNodeList[] la liste de E_nodes vivants qui est à tout moment trié en ordre croissant selon le cout du E_node, soit E_node_sol la solution au problème, alors on aura déjà parcouru tous les E_nodes qui ont un coût plus petit que le E_node_sol, et qui eux-mêmes ne sont pas une solution, car si ils étaient, on se serait déjà arrêté, et dans liveNodeList, on a les E_nodes qui ont peut être des childs réponse mais vu que liveListNodes est organisé selon le cout des E_nodes, alors le E_node_sol a forcément un coût plus petit que tous les E_nodes restants dans liveListNodes.

Vu que le cout(E_node) solution est le nombre d'étapes déjà fait, et que on a que cette valeur est donc plus petite que tout E_node dans liveNodeList, alors cet E_node_sol est forcément la solution optimale.

1.2)

Cf fichier python CostadaQuintaJoaoFilipe.py

Ce n'était pas clair si on devait implémenter la fonction du cours, mais le fait qu'on n'ait pas certaines fonctions nécessaires (par exemple addToLiveListNode()), et vu qu'on devait faire que solve() et c_hat() il était impossible de respecter le code généraliste, j'ai néanmoins essayé d'y rester proche.

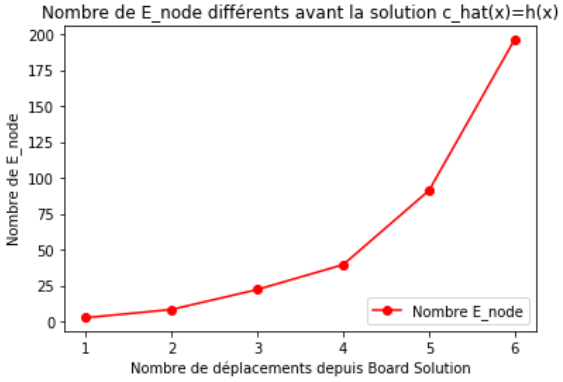
1.3)

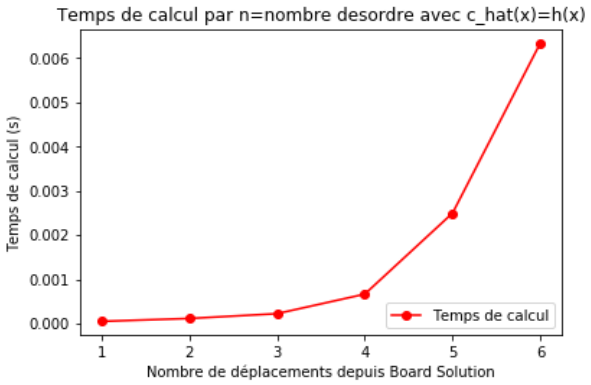
1)

Dans cet exemple c_hat(x) = h(x) :

Ce qu'on fait c'est une exploration de tous les nœuds en parallèle, imaginons que le premier nœud, le nœud 0, a 4 enfants, [1,2,3,4] ils ont tous un cout = 1, c'est leur depth, on va d'abord prendre le nœud 1, qui a au maximum 3 enfants, [5,6,7] qui on un coût = 2, alors on retourne au nœud 2, qui a aussi maximum 3 enfants [8,9,10] qui coûtent aussi 2, alors on revient encore au nœud 3 et si de suite jusqu'à avoir trouvé une solution. C'est donc une exploration en largeur.

N = [1, 2, 3, 4, 5, 6]
F(N) = [3.0, 8.7, 22.6, 39.9, 91.5, 196.7]





Graph (1) et (2)

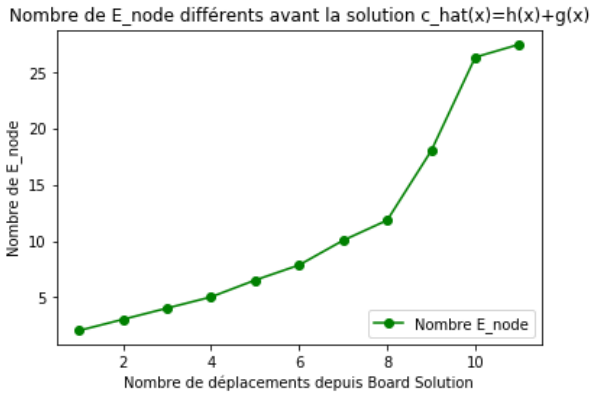
2)

Ici on a c_hat(x) = h(x) + g(x) :

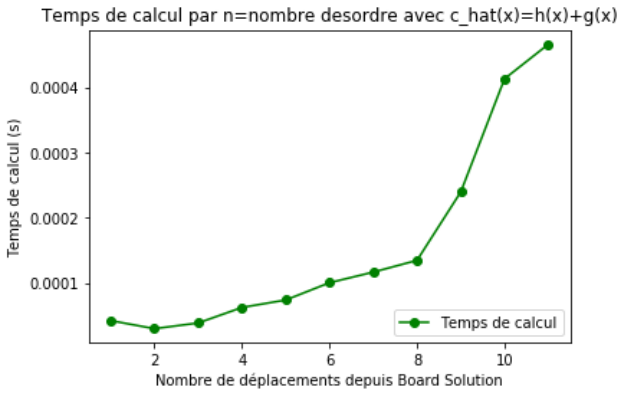
En regardant les graphiques on voit clairement une différence dans le temps de calcul ainsi que dans le nombre de E_nodes choisis, l'utilité d'une fonction cout bien défini est donc très visible

N = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

F(N) = [2.0, 3.0, 4.05, 5.1, 6.15, 8.25, 10.35, 11.85, 15.65, 19.95, 25.65]



Graph (3)



et (4)

La fonction solve(board) prend le premier E_node = board, on vérifie s'il est une solution, si non on prend ses enfants, maximum 4, on leur assigne un cout, et le nouveau E_node est l'enfant le moins chère qui aura lui au maximum 3 enfants, et si de suite.

Soit le board d'appel à solve = gen_disorder(board_résolu, n), où n est le nombre de déplacements depuis board_résolu, alors le calcul :

complxité = 4

while n > 1 :

complexité = complexité * 3

n = n - 1

complexité est alors égal au nombre max de E_nodes pour cet algorithme.

Pour n allant de 1 à 11, alors le nombre max de E_nodes est :

calculs_max = [4, 12, 36, 108, 324, 972, 2916, 8748, 26244, 78732, 236196]

Pour chaque E_node regarde s'il est une solution avec is_solution(E_node), soit O(is_solution) la complexité de cette fonction, le nombre de E_node maximum est aussi le numéro maximum de enfants, pour chaque enfant on calcule son cout avec la fonction c_hat(x), qui a la complexité O(c_hat) = O(count_bad_cells), finalement on doit insérer tout child dans liveNodeList en ordre croissante dans le pire des cas chaque child sera plus chère que tous les childs précédents, il faut donc parcourir toute la liste, soit longueur = len(liveNodeList), alors la complexité de cet algorithme est le nombre max de E_nodes multiplié par O(c_hat) = O(count_bad_cells) multiplié par la longueur.

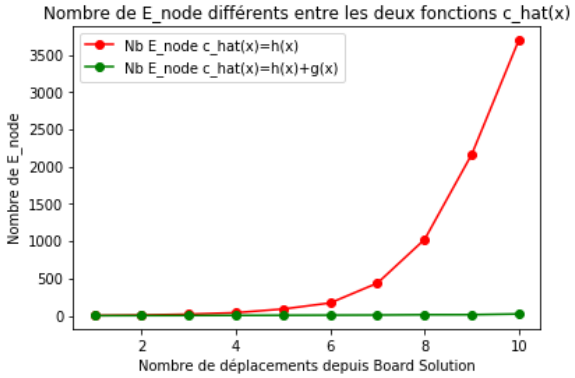
O(solve) = max_E_nodes * O(count_bad_cells) * longueur

Parcourir toutes les E_nodes correspond à l'algorithme avec c _hat(x) = h(x)

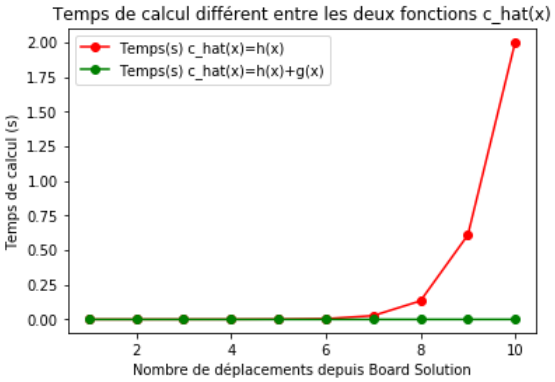
J'ai fait les mêmes calculs pour les deux fonctions différentes c_hat(x), avec n de 1 à 10, pour avoir un temps de calcul raisonnable, et comblé les graphiques.

En rouge on a la « mauvaise » version de c_hat(x), qui correspond à la complexité théorique du pire cas de l'algorithme, en vert on la « bonne » version de c_hat(x) qui correspond à la complexité pratique de l'algorithme.

Les graphiques (5) et (6) montrent la monstrueuse différence de E_nodes différents choisis et le temps de calcul.



Graph (5)



(6)