Rapport final du Projet Structure de données

TEYEB Nidhal
3802668

CANITROT Julien
3803521

Introduction

Ce projet présente différentes structures informatiques permettant de résoudre le Jeu Flood-it. Les paramètres pertinents pour évaluer les algorithmes de résolution de ce jeu sont le temps de résolution d'une part et le nombre de coups d'autre part. En première partie, nous nous intéresserons à un moyen de résoudre le jeu par des algorithmes utilisant une démarche aléatoire. Pour ces algorithmes, le nombre de coups nécessaires pour résoudre la zone de jeu étant difficilement optimisable, nous nous concentrerons essentiellement sur la vitesse d'exécution. En deuxième partie, nous décrirons l'implémentation de fonctions plus intelligentes dont le but est de résoudre le jeu en un nombre de coups beaucoup moins important. Dans cette partie, nous évaluerons aussi bien la vitesse d'exécution que le nombre de coups.

Partie 1

Exercice 1

La première fonction utilise la récursivité pour résoudre le jeu. A chaque tour, on part de la case supérieur gauche (i=0 / j=0), en récupérant sa couleur *cl*, puis on cherche toutes les cases adjacentes qui sont de la même couleur et on les marque. Pour chaque case ainsi marquée, on cherche à nouveau les case adjacentes non marquées de couleur *cl* jusqu'à ne plus trouver de nouvelles cases non marquées (récursivité). On choisit alors aléatoirement une couleur différente de celle de la case initiale et on attribue cette couleur à toutes les cases marquées. On répète ainsi cette boucle jusqu'à finir le jeu.

Q 1.3: Pour obtenir des évaluations représentatives du comportement de la première fonction (paramètre exercice = 0), nous l'avons lancé plusieurs fois pour chaque entrée <taille><nombreCouleur><difficulté><graine><exercice><affichage>, en conservant le nombre de couleurs constant à 10 et en testant 3 valeurs de taille (10, 100 et 300), et 2 valeurs de difficulté (0 et 1). Ci-dessous, les sorties (→) nombre d'essais (coups) et temps associés à chaque lancement de la fonction.

10 10 0 5 0 0 →	nombre d'essais: 53, 98, 63, 86, 85,102 temps: 0.001067, 0.004473, 0.002265, 0.002106, 0.004054
10 10 1 5 0 0 →	nombre d'essais: 90,63,83,104,77 temps associé 0.003397, 0.002277, 0.002150, 0.004236, 0.001470
200 10 1 5 0 0 →	nombre d'essais: 503, 515, 498, 584, 534 temps: 0.563219, 0.604015, 0.573350, 0.841772, 0.431469
200 10 0 5 0 0 →	nombre d'essais: 1035,1024, 1076, 1060,1030 temps: 1.208234, 1.197278, 1.304111, 1.350268,0.969420
300 10 1 5 0 0 →	nombre d'essais: 515, 527,491, 512, 507 temps: 1.340159, 1.484067, 1.629493, 1.321890, 1.153430

300 10 0 5 0 0 → nombre d'essais: 1507, 1574, 1552, 1558, 1579
temps: 3.999839, 4.492295, 4.129839, 4.348518,3.967961

Pour contrôler le temps lié à l'affichage, nous avons lancé 3 fois la fonction en conservant les derniers paramètres d'entrée, et en changeant uniquement le paramètre affichage (de 0 à 1)

300 10 0 5 0 1 → nombre d'essais: 1535,1521,1559
temps: 15.691593,16.384197,16.563896

Plus la taille est grande, plus le nombre d'essais augmente. Au contraire plus le paramètre difficulté « augmente » plus le nombre d'essaie diminue. En effet, on peut vérifier avec l'affichage que lorsque le paramètre difficulté augmente, les cases de même couleur sont plus regroupées. On ajoute donc à la Zone plus de cases à chaque essai. Il est aussi intéressant de remarquer que la difficulté a d'autant plus d'impact que la taille est grande. Ainsi, pour une taille de 10, le nombre d'essais est similaire entre des difficultés 0 et 1 alors qu'il double entre difficulté 1 et difficulté 0 pour une taille de 200 et triple pour une taille 300.

En ce qui concerne le temps de calcul, d'après nos résultats, il dépend de la taille de la grille à résoudre et de la difficulté, comme c'était le cas pour le nombre d'essais. Dans cette fonction de résolution aléatoire, on s'attendrait d'ailleurs à ce que, à difficulté donnée, le temps de résolution soit complètement lié au nombre d'essais effectués. En pratique, on se rend compte que ce n'est pas le cas. Ainsi pour les entrées 200 10 1 5 0 0, on a un résultat en 515 essais pour un temps de 0.604015 et un résultat en 534 (+20) essais pour un temps de 0.431469 (-0.2) et donc une variation opposée à l'attendu. On suppose que cette variation est due aux performances de l'ordinateur.
On remarque que l'affichage fait aussi baisser la vitesse d'exécution ce qui est logique vu que nous somme obligé de remplir la grille d'affichage de la bonne couleur à chaque fois.

Enfin, lorsque l'on effectue les tests pour une grande dimension (aux alentours de 323), on aboutit à une erreur de segmentation. On peut en déduire que le nombre d'appels d'une fonction récursive est limité.

Exercice 2

Q 2.2 Cet algorithme fonctionne de la même manière que son homologue récursif à ceci près qu'on remplace l'appel récursif par une pile à laquelle on ajoute les cases.
Ci-dessous, les sorties (→) nombre d'essais (coups) et temps associés à chaque lancement de la fonction.

10 10 1 5 1 0 → nombre d'essais: 76, 65, 95, 85, 107
temps: 0.000207, 0.000765, 0.001504, 0.001403, 0.001316

10 10 0 5 1 0 → nombre d'essais: 91, 63, 98, 65, 79
temps: 0.000345, 0.001621, 0.002022, 0.001196, 0.000320

100 10 1 5 1 0 → nombre d'essais: 549, 512, 578, 553, 506
temps: 0.209486,0.221131,0.262887,0.231842,0.240779,

100 10 0 5 1 → nombre d'essais: 532, 589, 773, 788, 821
temps 0.218765,0.257258

200 10 1 5 1 0 → nombre d'essais: 509, 513, 551, 530, 506
temps: 0.923253, 0.897340, 1.093784, 1.033467, 0.967765

200 10 0 5 1 0 → nombre d'essais: 999, 1026, 1078, 1025, 1020
temps: 1.684441, 2.441382, 1.925469, 2.037896, 1.934145

300 10 1 5 1 0 → nombre d'essais: 524, 516, 516, 532, 524
temps: 5.630349, 4.886284, 4.903149, 5.241662, 4.763863

300 10 0 5 1 0 → nombre d'essais: 1543, 1519, 1534, 1487, 1489
temps: 15.729155 ,13.882585, 14.7525422, 13.789123, 15.233611

400 10 1 5 1 0 → nombre d'essais: 552, 559
temps: 13.726997, 12.904614

400 10 0 5 1 0 → nombre d'essais: 2105, 2088
temps: 53.139465, 50.807144

La situation est similaire à l'exercice précédent en ce qui concerne le nombre d'essais. En revanche, on remarque qu'avec cet algorithme la taille n'est ici plus limitée. Nous pouvons donc traiter des grilles plus grandes. Cependant, la vitesse en pâtit et c'est le temps de calcul qui devient limitant pour les grilles de grande taille.

Exercice 3

Cette fonction recherche également les cases adjacentes de même couleur mais la structure utilise un marquage particulier pour les cases de la zone inondée d'une part et les cases de sa bordure d'autre part. Ce marquage permet à la fonction de ne jamais reconsidérer des cases, déjà identifiées comme appartenant à la zone inondée.

Q 3.4 Cette structure est de loin la meilleure des 3 que nous avons testées jusqu'à présent. En effet, comme on peut l'observer, le temps d'exécution du programme pour cette structure est bien inférieur à celui des 2 précédentes et elle n'a pas de limite de taille non plus.
De plus nous l'avons amélioré, d'une part, en imposant à la fonction principale de sélectionner les couleurs à chaque essai parmi celles se situant à la bordure de la zone inondée, ce qui explique sa meilleure performance en nombre d'essais, et d'autre part en ne changeant pas les couleurs des cases de la zone à chaque tour de boucle, lorsque nous n'effectuons pas d'affichage. En effet la structure permet de déterminer à tout moment si une case donnée se situe dans la Zone, en bordure, ou dans aucune des deux. Ceci explique les bonnes performances en temps d'exécution du programme.

10 10 1 5 2 0 → nombre d'essais: 31, 43, 35, 32, 37
temps: 0.000059,0.000151,0.000196, 0.000055, 0.000181

10 10 0 5 2 0 → nombre d'essais: 39, 36, 32, 35, 38
temps: 0.000048, 0.000140, 0.000054, 0.000146, 0.000123

200 10 1 5 2 0 → nombre d'essais: 492, 478, 496, 491, 478
temps: 0.013420, 0.033452, 0.009900, 0.019759, 0.021839

200 10 0 5 2 0 → nombre d'essais: 1003,1045, 914, 948, 1020
temps: 0.009543, 0.041527, 0.022932, 0.020579, 0.024919

300 10 1 5 2 0 → nombre d'essais: 453, 484, 467, 485, 489
temps: 0.028407, 0.030174, 0.029756, 0.028942, 0.029183

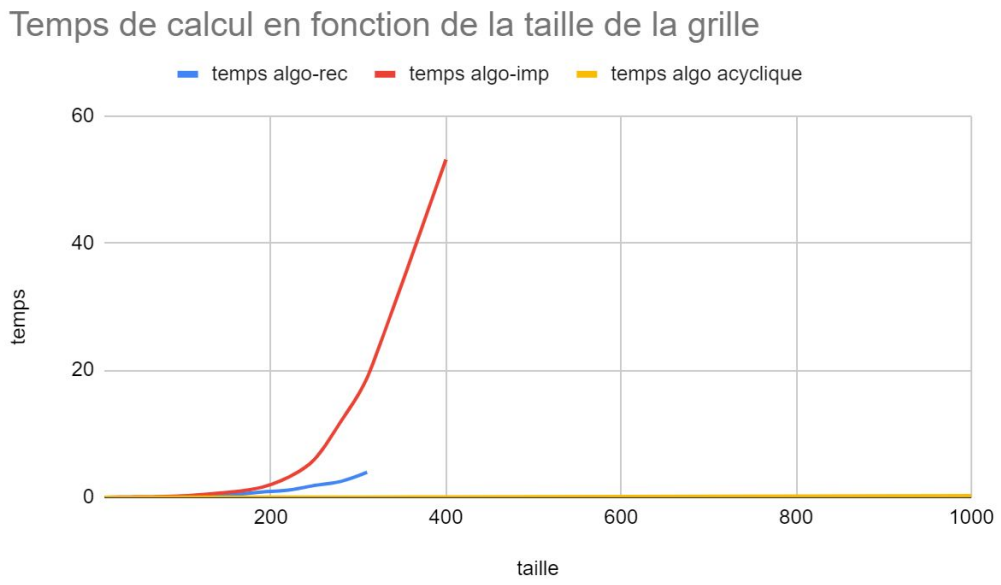
300 10 0 5 2 0 → nombre d'essais: 1486, 1528, 1489, 1479, 1467
temps: 0.028650, 0.032746, 0.036535, 0.039899, 0.029388

400 10 1 5 2 0 → nombre d'essais: 537, 521
temps: 0.044636, 0.051025

400 10 0 5 2 0 → nombre d'essais: 1987, 2068
temps: 0.054968, 0.048893
1000 10 1 5 2 0 → nombre d'essais: 532, 554, 566, 555, 553
temps: 0.230419, 0.235842, 0.221994, 0.234243, 0.236884
1000 10 0 5 2 0 → nombre d'essais: 4909, 4989, 4989, 4871, 4914
temps: 0.247498, 0.256900, 0.248019, 0.245049, 0.271771
10000 10 0 5 2 0 → nombre d'essais: 49073
temps: 55.522118

Ces résultats confirment que l'impact de la difficulté joue énormément sur le nombre de coups requis pour finir la grille et ce d'autant plus que la grille est grande.

Q 3.5 Le schéma ci-dessous résume les principaux résultats des 3 algorithmes. La courbe du temps de calcul pour l'algorithme récursif (en bleu) est interrompue par une limitation dans la profondeur des appels récursifs. Nous avons stoppé la courbe du temps de calcul de l'algorithme impératif (en rouge) par un seuil de temps de 50 sec. Ce seuil n'est atteint par l'algorithme acyclique que pour une taille de 10 000 (non représenté sur le schéma).



Partie 2

Exercice 4

Dans cet exercice, on crée une structure en graphe permettant de déterminer 1) des sommets qui regroupent les cases de même couleur adjacentes entre elles, et 2) les arêtes qui relient des sommets adjacents de couleurs différentes?

La fonction principale de cet exercice, *cree_graphe_zone*, a pour office de créer le graphe tout entier, et suit un principe simple : ”créer d’abord les sommets sans les arêtes, puis ensuite créer les arêtes”.

Exercice 5

Dans cet exercice, on utilise la structure de graphe, créée dans l’exercice précédent, ainsi qu’une nouvelle structure. Celle-ci se sert du principe d’un tableau de liste représentant les bordures de chaque couleur. Ce tableau ‘bordure’ représente les sommets adjacents à la zone inondée, répartis par couleur.

La fonction principale de cette exercice, *strategie_max_B*, consiste à choisir la couleur de la bordure la plus rentable, c’est à dire la bordure contenant le plus de sommets. Elle met à jour la zone inondée ainsi que sa bordure, à la manière de l’algorithme basé sur la structure acyclique de la partie 1.

Exercice 6

Dans cet exercice, on utilise les mêmes structures que pour la fonction *strategie_max_B*. On fait d’ailleurs appel à cette fonction également.

Le principe de la stratégie de cet exercice, s’appuie sur un parcours en largeur. Celui ci est effectué par la fonction *parcours_largeur*, appelée avec les coordonnées de deux cases, qui permet de déterminer le chemin le plus court entre ces deux cases.
Ce résultat est traduit dans la fonction *application_parcours_L*, par une suite de couleurs, qui seront utilisées pour inonder la zone, et donc atteindre la case voulue en un minimum de coups.

Une fois l’application du parcours en profondeur effectuée, on applique la fonction de la stratégie max bordure afin de terminer le jeu.
Toutes ces fonctions étant génériques, on peut alors les utiliser en choisissant les coordonnées, ce que nous avons fait pour les stratégies bonus.

Exercice Bonus

Après avoir créé des fonctions de parcours génériques dans l’exercice précédent, nous avons pensé à appliquer plusieurs parcours afin de réduire le travail de la fonction de la stratégie max bordure.
Nous avons alors effectuée deux fonctions, représentant deux stratégies différentes :

- une stratégie effectuant deux parcours, entre deux coins de la matrice.
- une stratégie effectuant trois parcours, entre trois coins de la matrice.

Ces deux stratégies donnaient de meilleurs résultats en nombre de coups, que la stratégie max bordure, mais bien inférieur au parcours en largeur tant bien en nombre de coup, que en temps d'exécution.
Ainsi nous avons conclu que multiplier le nombre de parcours n'améliore pas le nombre de coups, et ralenti grandement le temps d'exécution, alors nous avons décidé de ne pas analyser les résultats obtenues

Analyse statistique et graphes

Après l'implémentation de toutes les fonctions permettant de résoudre la grille, nous procédons à l'analyse des performances de chacune en considérant 1) le temps total de résolution et 2) le nombre de coups requis pour résoudre la matrice de jeu.

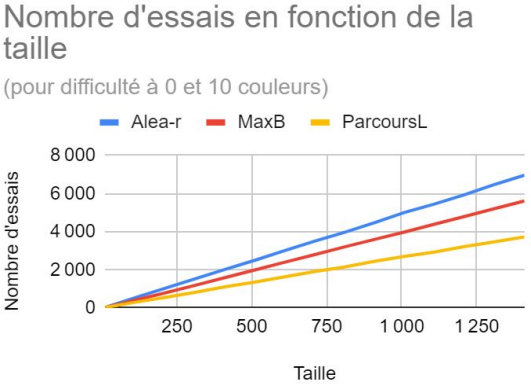
Afin d'obtenir une évaluation représentative des performances de chaque fonction dans différentes conditions, nous devons effectuer un nombre de test très élevé (afin de respecter une rigueur scientifique).
En effet, le protocole mis en place ici nécessite de faire varier chacun des 3 paramètres, soit la taille, la difficulté et le nombre de couleurs, indépendamment les uns des autres pour éliminer les facteurs confondants. Autrement dit, on a fait varier chaque paramètre que l'on voulait étudier en gardant les 2 autres fixés. Pour chaque combinaison de paramètre, nous avons de plus effectué des tests avec 15 graines différentes afin d'éliminer des facteurs aléatoires et d'obtenir une valeur plus représentative de la performance aux paramètres d'intérêts en calculant une valeur moyenne.
Pour effectuer ces tests, nous avons ajouté, directement dans le main, des lignes de code qui implémentent le protocole ci-dessus sous forme de boucles sur chacun des paramètres.

Cette approche nous permet d'observer l'effet de chaque paramètre sur le temps de calcul et le nombre de coups, les deux autres paramètres étant fixés. En revanche, nous n'avons pas observé systématiquement les interactions entre paramètres. Par exemple: l'effet de taille est-il d'autant plus important que la difficulté augmente ? L'analyse de ces interactions nécessiterait beaucoup plus de données. Nous avons tout de même analysé une interaction entre les paramètres de taille et de difficulté qui nous semblait intéressante car nous nous pensions l'avoir observée dans la partie 1.

A partir des données ainsi recueillies, nous avons essayé de résumer les résultats sous la forme de graphes qui permettent de comparer visuellement les 3 stratégies de résolution qui nous intéressent le plus, c'est-à-dire la plus rapide des aléatoires (structure acyclique), la max bordure et le parcours en largeur. Toutes les données produites et utilisées pour les graphiques sont fournies dans les feuilles de calcul disposées dans le fichier tar.gz rendu)

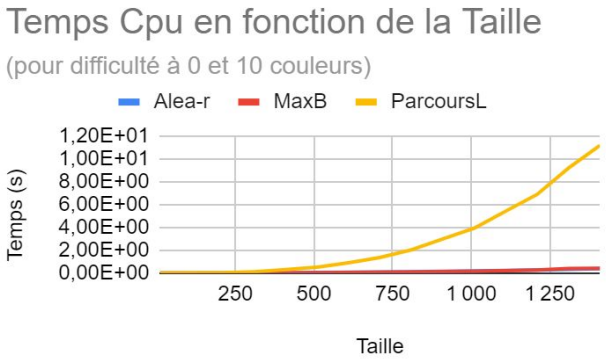
Effet de la taille

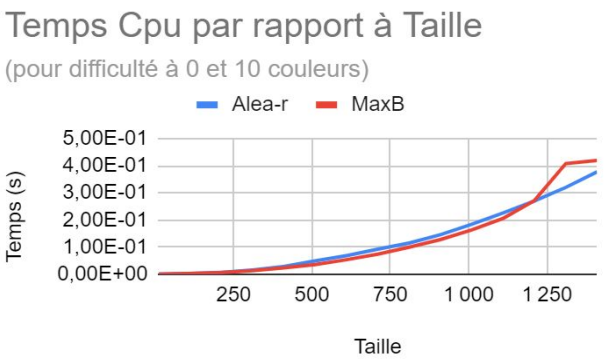
On peut observer sur ce premier graphe que le nombre d'essais nécessaire pour résoudre une grille dépend linéairement de sa taille, quelle que soit la stratégie utilisée. En revanche, la pente dépend de la stratégie, ce qui signifie que les stratégies ont des efficacités relatives différentes. Ainsi le parcours en largeur est presque 2 fois



plus efficace que l'aléatoire. L'efficacité de la fonction max bordure se situe entre les deux.

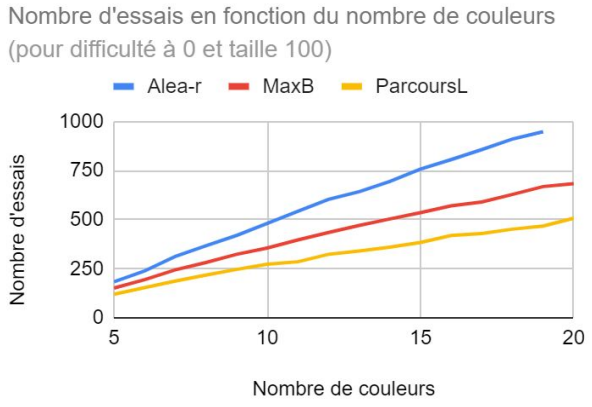
L'impact de la taille sur le temps est très différent de l'impact sur le nombre d'essais: il n'est plus linéaire. En effet, on peut d'un premier abord voir que l'efficacité de ce parcours en largeur est ici bien moindre (schéma ci-dessous à gauche). Nous avons supprimé la courbe ParcoursL sur le schéma de droite pour mieux observer l'allure des deux autres fonctions. Ainsi on peut observer que l'allure de l'évolution pour les 3 fonctions est plutôt polynomiale (voir schéma en échelle log sur la feuille de calcul "Affichage LOG").

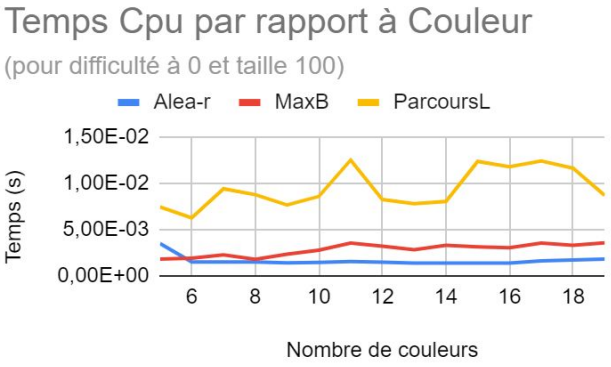




Effet du nombre de couleurs

On peut voir que, comme pour la taille, le nombre d'essais (graphe ci-dessous à gauche) semble augmenter de façon linéaire avec le nombre de couleurs. On remarque que l'ordre d'efficacité des algorithmes est le même: Parcours Largeur est plus efficace que MaxBordure, lui-même plus efficace que Aléatoire.

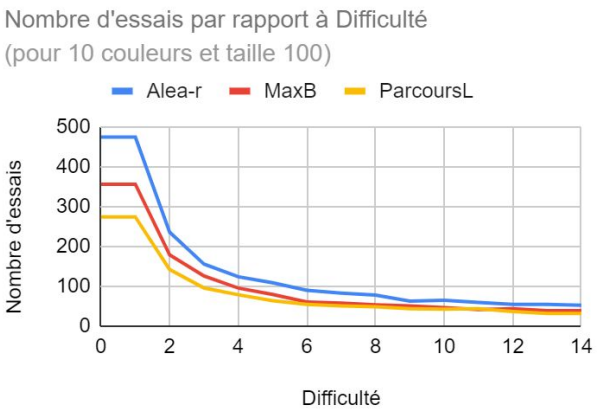


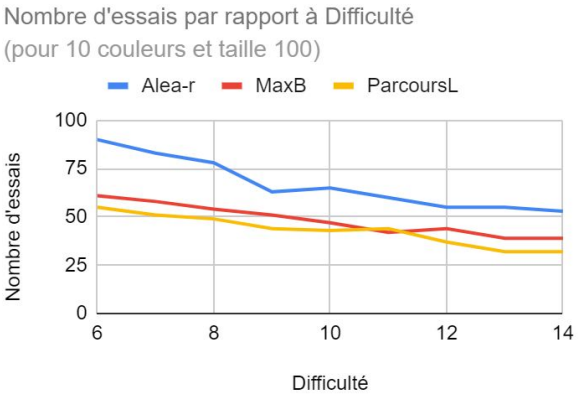


Le nombre de couleurs, en revanche, ne semble pas impacter le temps de résolution qui apparaît constant (schéma ci-dessus à droite). On remarque que les algorithmes conservent leur classement.

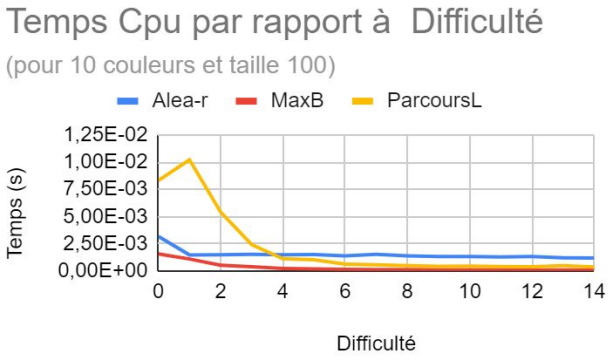
Effet de la difficulté

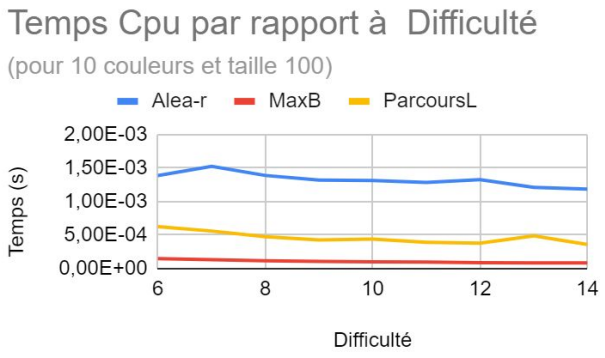
Le paramètre difficulté fait décroître le nombre d'essais. L'impact de ce paramètre est de moins en moins important à mesure qu'on augmente la difficulté. De plus, la différence entre les algorithmes est de moins en moins importante à mesure qu'on augmente le paramètre difficulté (voir zoom sur les valeurs 6 à 14, schéma ci-dessous à droite).





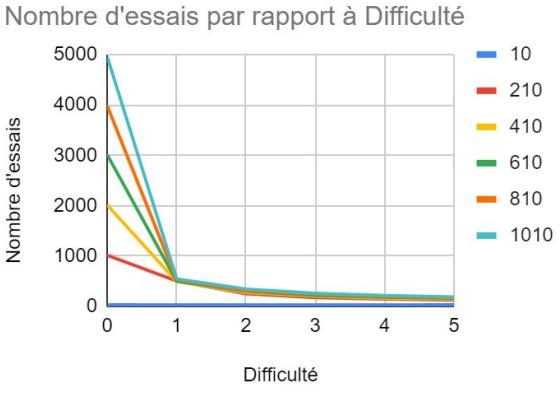
Le temps cpu semble ne pas varier pour les difficultés au-delà de 6 (voir schéma de droite ci-dessous et à mettre en relation avec l'allure des courbes nombre d'essais ci-dessus). On peut tout de même remarquer que pour une difficulté 0 le temps semble légèrement plus long, surtout pour le parcours en largeur (schéma de gauche ci-dessous). Cependant au vu de ces résultats nous ne pouvons pas conclure à une corrélation entre temps cpu et difficulté.



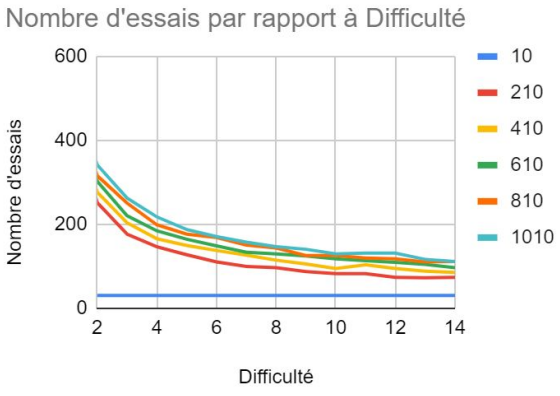


Interaction entre les différents paramètres

Les graphes ci-dessous sont un exemple possible d'analyse d'interaction entre différents facteurs d'entrée. Nous avons décidé d'analyser ici l'interaction entre difficulté est taille car dans la partie 1, nous avons observé que l'impact de la difficulté semblait d'autant plus grand que la taille augmentait. On retrouve ici que plus la taille est grande plus la difficulté diminue le nombre d'essais. Ceci est vrai pour le passage de difficulté 0 à 1, qui a un impact d'autant plus grand que la taille est grande.



Cependant, on observe dans le deuxième graphe que pour des difficultés plus grandes cette interaction entre taille et difficulté n'est plus visible. On se retrouve ainsi avec un nombre d'essais homogène entre les différentes tailles pour des difficultés strictement supérieures ou égales à 2. Une exception cependant, on observe



que pour une taille 10 la difficulté n'a aucun impact sur le nombre d'essai. En pratique il faut une difficulté beaucoup plus grande de l'ordre de 100 pour que la difficulté ait un effet sur la résolution des petites grilles.

Conclusion

On remarque à travers l'analyse des graphes des différents tests effectués, que le paramètre 'taille' influe sur le nombre d'essais ainsi que sur le temps d'exécution du programme, alors que le nombre de couleurs et la difficulté n'ont d'incidence que sur le nombre d'essais.
De manière générale, l'algorithme utilisant la structure acyclique est plus efficace que le reste des algorithmes en temps d'exécution mais moins en nombre d'essai. C'est l'inverse pour la stratégie Parcours en largeur qui est plus efficace en nombre d'essais mais bien plus coûteuse en terme de temps. L'algorithme de la stratégie 'max bordure' relève, quant à lui, d'un bon compromis au vu de ses performances en temps et en nombre d'essais.

Nos résultats donnent un bon aperçu des qualité spécifiques de chaque algorithme, ainsi que de l'effet de chaque paramètre sur la résolution de la grille. Cependant, cette approche ne permet pas d'analyser des effets croisés entre paramètres comme celui que nous avons noté entre taille et difficulté. Il serait intéressant d'étudier plus systématiquement les interactions entre tous les paramètres.

Nous avons observé que la réduction des fuites de mémoire améliorait la performance du programme en terme de temps d'exécution. Une grosse partie de notre temps de travail a donc été consacrée à débusquer les fuites de mémoire et à optimiser le code. Cependant, quelques petites fuites de mémoire persistent encore. Il aurait peut-être été plus facile de les éradiquer si nous nous étions concentré sur les fonctions de libération à chaque implémentation d'un nouvel algorithme plutôt qu'en fin de projet, une fois que l'ensemble fonctionnait.

Pour la réalisation des statistiques, nous avons programmé des boucles qui effectuent un grand nombre de tests dans le programme principal (avec l'option exo=6). Nous aurions pu réaliser ces boucles dans un fichier à part, et en utilisant des fonctions auxiliaires génériques, afin de faciliter la visibilité au correcteur et à tout autre personne voulant lire notre code.

Ce projet nous a permis de nous améliorer en tant que développeur, car étant données les conditions dans lesquelles nous étions, nous avons réglé nos problèmes tout seuls. Nous avons mis fin aux quelques zones d'ombres qu'il nous restait, sur les pointeurs par exemple, et avons amélioré notre compréhension de l'outil valgrind.
En conclusion, nous avons tiré beaucoup de points positifs de ce projet, et paradoxalement des conditions dans lesquels nous, comme l'ensemble du pays, nous trouvions. Le fait d'être relativement livrés à nous même en tant que jeunes développeurs, nous a appris à apprendre seuls et fait gagner en autonomie.