Projet : Flood It

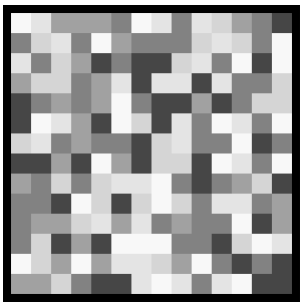
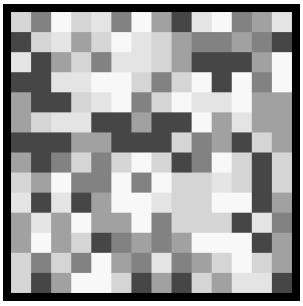
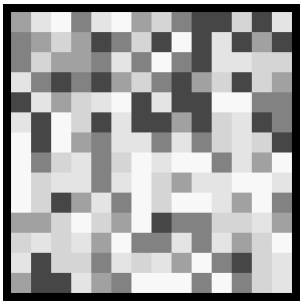
Etude de stratégies algorithmiques

Matthieu de Castelbajac

Abstract

Nous nous intéresserons ici à la résolution du jeu d'inondation **Flood It**. Une grille de jeu présente plusieurs zones colorées. L'objectif : inonder la grille avec une seule couleur, en modifiant la couleur de la zone supérieure gauche, la *ZSG*, pour l'agrandir. On appellera zone un ensemble de cases adjacentes de même couleur. Nous implémenterons différentes stratégies de résolution du jeu et étudierons leur efficacité selon deux paramètres, le *temps d'exécution* et le *nombre de coups*, en nous demandant si une seule stratégie peut optimiser ces deux paramètres. Dans un premier temps, nous étudierons plusieurs implémentations d'une stratégie de sélection aléatoire des couleurs en modélisant le jeu par une matrice. Dans un second, nous tenterons d'implémenter des stratégies minimisant le nombre de coups en modélisant le jeu par un graphe.

Le présent rapport s'organise comme suit : nous détaillerons l'organisation du code et son exécution par le lecteur. Nous discuterons ensuite des différentes stratégies et de leur(s) implémentations. Puis nous les comparerons selon les deux critères évoqués plus haut.

Manuel d'utilisation

L'archive fournie contient deux dossiers ***'Partie_1'*** et ***'Partie_2'***. Le premier renferme le code relatif au modèle "matrice". Le second, celui relatif au modèle "graphe".
Chaque dossier contient des fichiers **'.c'**, les ***headers*** correspondants qui fournirons un courte description de chaque fonction, un *Makefile* pour la compilation, et un fichier *test.c* qui contiendra les jeux de tests de chaque partie.

Partie_1
Les fichiers API permettent un affichage graphique et sont à ignorer. La première stratégie se décline en trois implémentations (*Fonctions_exo1,–exo2,–exo3*) mais nous y reviendrons. Le fichier *Flood-It_Partie1.c* permettra de les exécuter. Le lecteur pourra d'ailleurs y parvenir en utilisant la commande suivante dans un terminal :
./Flood-It_Partie1 <largeur de la grille> <nombre de couleurs> <difficulté> <graine> <numero exercice> <affichage graphique>
Où la largeur est exprimée en nombre de cases, la difficulté représente le % de chance d'ouvrir le jeu avec de plus grandes zones et où la graine est un entier permettant de sauvegarder la configuration initiale d'une grille. Elle nous servira grandement par la suite.

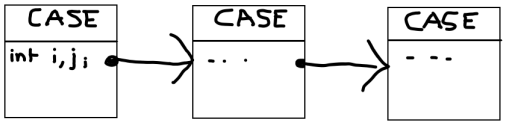
Partie_2
Toutes les fonctions communes sont trouvables dans *Fonctions_graphe.c*. Trois stratégies sont implémentées. La Première dans *Max_bordure* et les deux autres dans *Parcours_en_largeur.c*. Le programme pourra être exécuté de façon analogue, mais sans préciser le champ binaire sur l'affichage graphique, par la commande :
./Flood-It_Partie2 <largeur de la grille> <nombre de couleurs> <difficulté> <graine> <numero exercice>

Pour faire tourner les jeux d'essais, il suffit d'exécuter la commande suivante :
./Test <numero exercice> < numero du test >
Deux ou trois tests sont disponibles, selon le répertoire. Les deux premiers permettent de faire varier respectivement la taille de la grille et le nombre de couleur. Dans le second répertoire uniquement, le troisième permet de faire varier la difficulté. Toutes les constantes utilisées peuvent être modifié *via* les **#define**.

Sélection aléatoire des couleurs

Modélisation

La grille de jeu est modélisée par une matrice d'entier qui indiqueront la couleur de la case correspondante.
Nous chercherons à optimiser un algorithme de sélection aléatoire des couleurs. Cet algorithme pourra comporter les fonctions suivantes :
• **trouve_zone()** : Elle permet de stocker les cases d'une zone donnée sous forme de liste simplement chaînée et d'en retourner (valeur de retour ou passage par pointeur) la longueur. La structure est de fait adaptée puisque la taille (les cases) de la zone est inconnue jusqu'à ce qu'elle soit entièrement parcourue.
Elle nous servira en particulier a définir, puis agrandir la *ZSG*.
• **sequence_aleatoire()** : C'est la fonction qui résout le jeu par l'intermédiaire de **trouve_zone** et renvoie le *nombre de coups* effectués jusqu'à la validation d'une **condition de victoire – ou de fin de jeu**, susceptible de varier selon la stratégie.



A chaque couple *(i, j)* est associé un entier dans la matrice, représentant la couleur de la case

Stratégie récursive

Commençons par une approche purement récursive.
D'une part, **trouve_zone** construit notre *ZSG* récursivement, un appel par case. D'autre part, **sequence_aleatoire** est rappelée récursivement à chaque changement de couleur. La condition de victoire est simplement d'avoir une *ZSG* de la taille de la grille.
Cette stratégie, si elle est rapide, reste fortement limitée en ce que cette double récursion ne doit pas déborder du thread. Elle n'est donc plus viable à partir d'une certaine taille de grille – la taille dépendant de la machine.

Stratégie PILE

C'est l'équivalent "dérécursifié" de la première stratégie. La récursion est remplacée par un modèle de **pile** avec empilement et dépilement successifs. Dit autrement, une case ajoutée lors d'une itération est dépilée, retirée de la liste et on ajoute si besoin les cases qui lui sont adjacentes. La stratégie à l'avantage de ne pas recourir à la récursion. Elle peut donc *a priori* traiter des grilles de beaucoup plus grande taille. Cette stratégie devrait troquer ledit avantage contre un plus mauvais *temps d'exécution*.

Stratégie Bordure

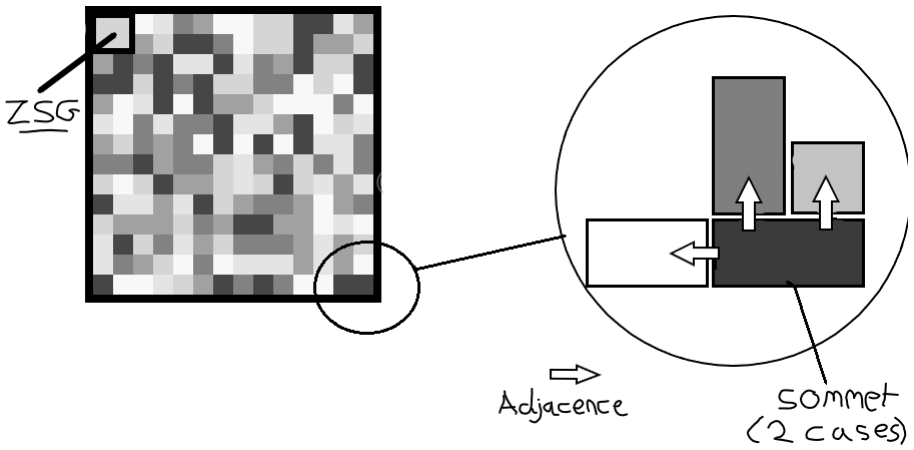
Jusqu'ici la *ZSG* était recalculée à chaque coups. Pour y remédier, la *ZSG* est maintenant stockée sous forme de liste, à part. De même, sa *Bordure – ie* les cases adjacentes à la *ZSG* – est stockée sous forme de tableaux de listes. Toutes les cases sont réparties selon leur couleur, afin de pouvoir les ajouter rapidement.

Plus besoin de **trouve_zone** donc. Cependant la structure utilise une série de marqueurs pour discriminer rapidement une case donnée. Une nouvelle matrice est ajoutée. Elle ne contient non pas les couleurs, mais l'appartenance des cases (*ZSG, Bordure, etc*).
De plus, nous aurions pu stocker la taille de notre *ZSG* et l'utiliser comme condition de victoire. Mais une autre condition valide permet de s'assurer de la fin d'une partie **sans** devoir parcourir une structure. En effet, il suffit que la *Bordure* contienne – nécessairement – au moins une case, et que durant le dernier coups, aucune case qui n'est pas encore dans la *ZSG* y soit ajoutée.
Ainsi si l'on ne s'occupe pas vraiment du *nombre de coups* pour l'instant, choisissant toujours les couleurs aléatoirement, pouvons-nous tout de même optimiser le temps d'exécution. En effet celui-ci dépend surtout des parcours de structures. En limitant de façon drastique les parcours – la *ZSG* n'est jamais reparcourue, la Bordure est directement triée selon la couleur grâce au tableau – nous devrions observer une diminution, non seulement significative, mais en plus importante du *temps d'exécution*.

Stratégies sur les graphes

Modélisation

Dans cette seconde partie, on choisit de modéliser la grille non plus par une matrice de cases, mais par un graphe non orienté, dont les sommets représentent les zones de couleur. De plus, on définira comme sommets adjacents deux sommets dont au moins une case de chaque est adjacente à l'autre. Chaque sommet pourra donc répertorier ses propres cases, son nombre de case, et ses sommets voisins. L'intérêt est ici que le graphe n'est déterminé qu'une seule fois en début de partie. Plus besoin, donc, de construire des listes de cases à chaque tour, les adjacences n'évoluant pas.
Nous reprendrons cependant la séparation du graphe selon la stratégie précédente, la *ZSG* où les sommets de la même couleur que celui contenant la case supérieure gauche, et la *Bordure*, sommets adjacents à la *ZSG*. Il faut par là comprendre que les sommets ne fusionnent pas pour former une *ZSG* mais que leur appartenance à celle-ci sera conditionnée par un entier.

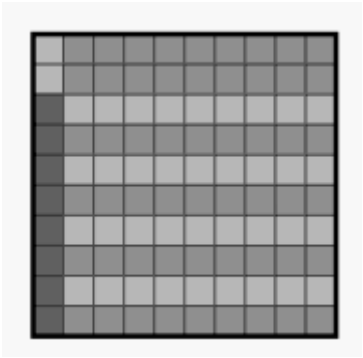


De ce modèle découlent deux stratégies :

Stratégie Max-bordure

Cela consiste à sélectionner la couleur du sommet de plus grande taille – *càd* contenant le plus de cases – parmi les sommets adjacents à la *ZSG*. L'intérêt viens de ce qu'un sommet de plus grande taille à plus de chance d'avoir beaucoup d'adjacents, ce qui devrait limiter le *nombre de coups*. Car c'est bien le paramètre que l'on cherche à diminuer ici – il nous faudra néanmoins comparer les *temps d'exécution* des différents algorithmes selon le modèle.
Son implémentation est quasi immédiate puisque, nous l'avons dit, chaque sommet stocke son nombre de cases. Il suffit alors de sélectionner la taille maximale parmi la liste des adjacents étranger à la *ZSG* d'un sommet de la *ZSG* – et de réitérer pour chaque sommet de la *ZSG*.
Remarquons que, si cette simple stratégie peut s'avérer redoutablement efficace, elle est aussi exécrable – en *nombre de coups* – dans certaines configurations de grille.

Voici un exemple [1] :



Il est assez évident que cet grille peut être résolue en 3 coups, par la séquence *gris foncé -> gris moyen -> gris clair*. Pourtant, la zone *gris foncé* est toujours de plus petite taille, portant le nombre de coups selon notre stratégie à 10. Notons au passage que même une sélection aléatoire des couleurs serait plus efficace.

Parcours en largeur

Une stratégie qui éviterait ce désagrément consiste à ajouter à la *ZSG*, selon le plus petit nombre de coups, le sommet inférieur droit, à l'autre bout de la grille, puis à utiliser la stratégie précédente. Pour cela, nous avons besoin de parcourir le graphe, en largeur, en attribuant à chaque sommet un sommet père, selon le plus court chemin vers la racine, soit notre *ZSG* de départ NOTE. Il suffit alors de sélectionner la séquence de couleur permettant d'accéder, toujours selon le plus court chemin, au sommet inférieur droit.

Nous nous attendons bien sûr à ce que ces deux stratégies soient de loin plus efficaces en nombre de coups que la stratégie de sélection aléatoire.
Reste à savoir si l'efficacité en temps d'exécution souffrira de la même comparaison.

Autres stratégies

Le parcours en largeur permet la formulation d'autre stratégies de résolution : plus court chemin vers les sommets les plus grands, les plus distants... De façon générale, ces stratégies permettent de calculer un certain nombre de coups à la fois. Ce calcul en "brute force" peut même être poussé à bout par calcul du plus petit *nombre de coups* possible pour une grille donnée par parcours récursif du graphe de liaison donné par le parcours en largeur. Bien entendu, cela vient au détriment du *temps d'exécution*. Si, à l'inverse, l'optimisation recherchée concerne le *temps d'exécution*, alors, nous l'avons dit, il faut minimiser le nombre de parcours des différentes structures.
Dans ce cadre, nous implémenterons une stratégie similaire à la précédente, à une différence prêt. Nous recueillons lors du parcours en largeur le sommet le plus distant et faisons en sorte de l'atteindre selon le plus court chemin. Dans les cas ou le sommet inférieur droit n'est pas le sommet le plus distant, nous devrions obtenir une légère amélioration du *nombre de coups*.

[1] Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, Benjamin Sach, **The complexity of flood filling games**, August 2010. voir PDF

Comparaison des stratégies

Nous procéderons aux jeux de tests suivants :

1) Dans un premier temps, nous comparerons l'efficacité en *temps d'exécution* de tous les algorithmes – le *nombre de coups* étant variable pour les trois premiers. Ce faisant nous fixerons une graine pour toute l'expérience, pour limiter l'influence de l'aléatoire, et – pour l'instant – une difficulté. Nous ferons alors varier la taille de la grille en fixant le nombre de couleur, puis inversement.

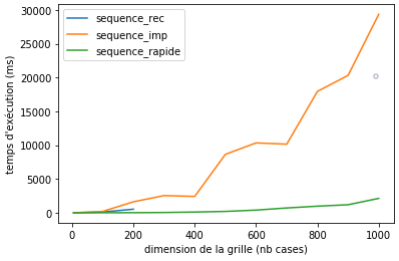
2) Dans un second nous nous concentrerons sur les algorithmes les plus performants en *nombre de coups* et en étudierons les performances selon les mêmes variations.

3) Enfin, nous illustrerons l'intérêt des deux dernières stratégies dans un cas précis, à savoir la résolution de grandes grilles avec une difficulté de faible pourcentage. En effet, la difficulté n'a a priori pas un grand impact sur les algorithmes aléatoires, mais joue considérablement sur le nombre de sommets.

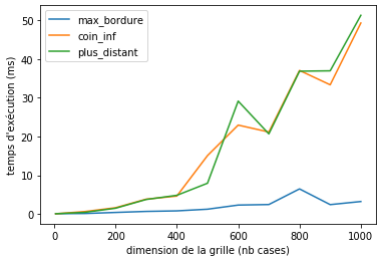
NB : Les tests ont étés réalisés sur une machine virtuelle, avec une puissance de calcul limitée. Nous invitons le lecteur a relancer ces tests sur sa machine s'il souhaite préciser leur analyse quantitative.

Taille de la grille et performances

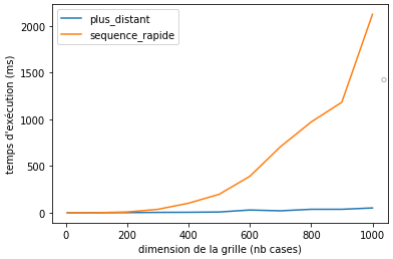
Observons dans un premier temps l'influence de la taille de la grille sur le *temps d'exécution*.
Fixons d'abord les paramètres suivant : *nbcl = 10 , diff = 33 , graine = 0*
La taille de ma grille prendra les valeurs 5, 10, puis des valeurs de 100 à 1000 avec un pas de 100.
Voici ce que nous obtenons :



(a) Séquence aléatoire



(b) Stratégies sur les graphes



(c) Comparaison

Remarquons en *(a)* que, comme attendu, l'algorithme récursif s'essouffle très rapidement, et que, de plus, passer du modèle *PILE* à une structure acyclique permet de réduire le temps d'exécution d'un facteur 10 pour les grandes grilles.

En *(b)*, l'algorithme le plus performant sur les graphes est sans surprise le premier : il effectue le moins de calcul.

Nous illustrons en *(c)* la disparité des valeurs recueillis pour les deux modèles. La conclusion est sans appel. Modéliser le jeu sous forme de graphe s'avère plus efficace.

Regardons donc maintenant l'influence de la taille de la grille sur le *nombre de coups*, cette fois. Bien entendu, cet aspect n'a d'intérêt que pour les stratégies sur les graphes :

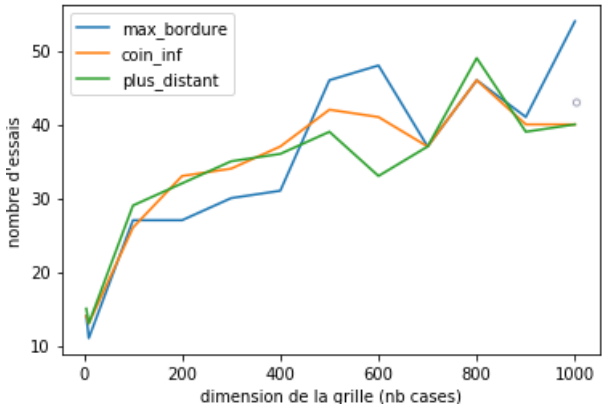
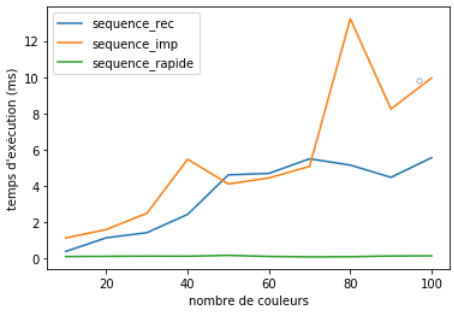


Figure 2: Nombre de coups en fonction de la taille de la grille

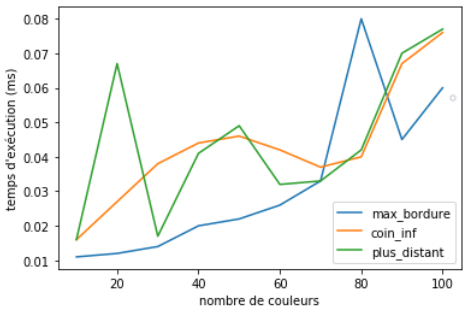
Malheureusement, le graphe nous en apprend peu. Aucune stratégie ne semble prouver son efficacité ou inefficacité systématiquement. Chacune s'illustre sur de petits intervalles et de façon anecdotique.

Nombre de couleurs et performances

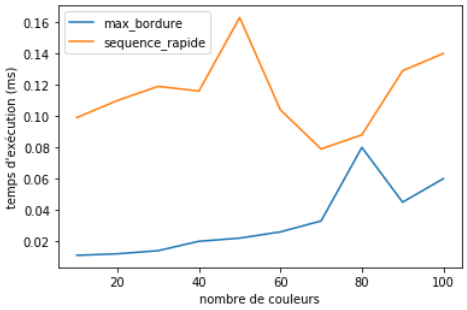
Observons maintenant l'influence du nombre de couleurs sur le *temps d'exécution*.
Fixons les paramètres qui doivent l'être : *dim = 10* , *diff = 33* , *graine = 0*
Le nombre de couleur *nbcl* prendra des valeurs de 10 à 100 avec un pas de 10.
Voici ce que nous obtenons :



(a) Séquence aléatoire



(b) Stratégies sur les graphes



(c) Comparaison

Figure 3: Temps d'exécution en fonction du nombre de couleur

En *(a)*, sans surprise ou presque, l'algorithme de résolution par sélection aléatoire des couleurs le plus efficace est le troisième, et le plus inefficace, le deuxième, toujours d'un facteur 10. Remarquons tout de même que le premier, s'il reste légèrement plus efficace que son équivalent 'dérecursifié', voit ses performances bien inférieure à la *sequence_rapide* ; réflexion que nous n'avions pas pu clairement formuler précédemment, compte tenu de ses limitations.

En *(b)*, de même que pour nos premières observations, *max_bordure* semble généralement plus efficace en *temps d'exécution*, mais les différences ne sont pas assez importantes pour être significatives.

Cependant, la comparaison en *(c)* nous livre une illustration plus précise de l'efficacité relative, et sur de petite grilles, de *sequence_rapide* par rapport aux stratégies basée sur les graphes.

Penchons-nous donc sur l'influence du nombre de couleurs sur le *nombre de coups*.

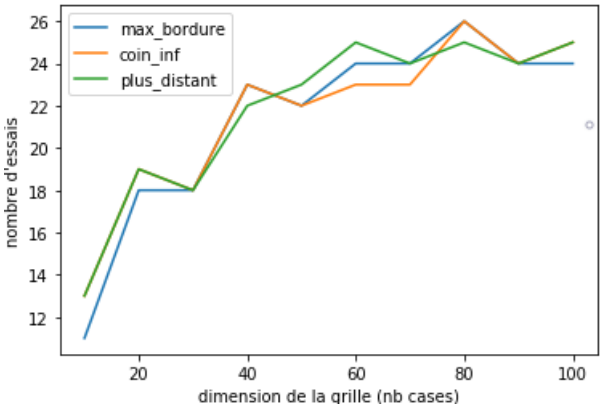
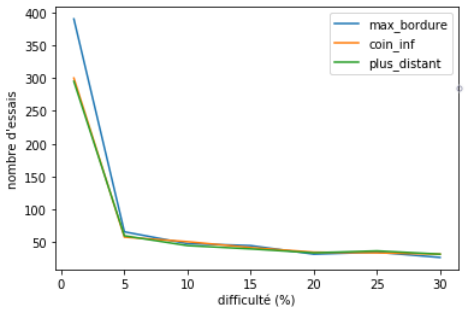


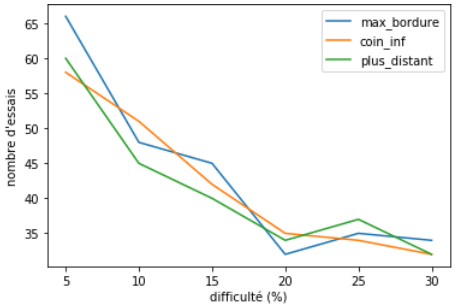
Figure 4: Nombre de coups en fonction du nombre de couleurs

L'augmentation du nombre de coup en fonction du nombre de couleurs suit une progression logarithmique, mais aucun algorithme ne s'illustre particulièrement.

Manifestement, ni la variation du nombre de couleurs ni celle de la taille de la grille ne permettent de départager nos trois algorithmes. Tournons-nous alors vers la difficulté. La difficulté, rappelons-le, s'exprime sous forme d'un pourcentage, celui d'obtenir à la création du jeu, deux cases adjacentes de même couleur. De fait, la difficulté détermine la taille des sommets d'une part et le nombre de sommets d'autre part. Le scénario qui nous intéresse ici concerne une grille de grande taille générée avec une faible difficulté.
Nous fixerons les paramètres suivants : *dim = 500 , nbcl = 10, graine = 0*
La difficulté prendra les valeurs suivantes : 1, puis des valeurs de 5 à 30 avec un pas de 5.
Voici les résultats obtenus représentés à deux échelles différentes :



(a) intervalle complet



(b) [5 : 30]

D'une part, le *nombre de coups* décroît selon une exponentielle inverse. D'autre part, pour de très faibles difficultés, *max_bordure* est moins efficace que les deux autres stratégies. Malheureusement, notre dernière stratégie ne réduit pas assez le *nombre de coups* dans ces configurations.

Conclusion

Des observations précédentes, nous pouvons tirer deux conclusions.

Le modèle selon lequel l'algorithme de résolution est implémenté – matrice ou graphe dans notre cas – détermine davantage son efficacité que l'algorithme lui-même. De fait, passer par un graphe s'avère redoutablement plus efficace en *temps d'exécution* que par une matrice, du moins dans la grande majorité des configurations (*cf* Figure 3). Concernant l'efficacité en *nombre de coups*, une bonne manière de s'assurer davantage de la prépondérance du modèle aurait été de comparer une stratégie *max-bordure* sur une matrice et une stratégie de sélection aléatoire des couleurs sur un graphe.

Mais plus important encore, nous n'avons pas pu formuler de stratégie qui combine *efficacement* nos deux paramètres. Dit autrement, aucune de nos stratégie parvient à résoudre le jeu selon le plus petit nombre de coups **et** le plus rapidement possible. Parmi les stratégies de sélection non-aléatoire des couleurs, la plus rapide requiert généralement le plus grand nombre de coups.

En effet, la résolution du jeu Flood-It selon le plus petit nombre de coups est considérée comme un problème **NP-difficile**[1]. Nous pouvons donc conclure que toute solution qui minimise le nombre de coup ne peut qu'approcher une résolution en temps polynomial – par rapport aux paramètres d'entrée – et ne peut donc pas être la plus rapide. Et inversement, toute solution qui minimise le temps de résolution contient contient des coups redondants et ne peut minimiser le nombre de coups.

Néanmoins, nous pouvons encore implémenter des algorithmes plus efficaces que ceux présentés ici selon l'un, ou bien l'autre paramètre.

[1] Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, Benjamin Sach, **The complexity of flood filling games**, August 2010. voir PDF