LU2IN006 - Compte rendu Flood-It

LOI Alessia, NAUMENKO Polina
Groupe 6

Ce projet s'intéresse à la recherche des stratégies pour gagner au jeu d'inondation de grilles. Le but est de recouvrir une grille carré d'une seule couleur, en partant d'une grille composée d'un certain nombre de couleurs, réparties aléatoirement sur la surface. Le projet est l'occasion pour étudier la structure des données graphes et leur coloration.

Structures manipulées dans notre projet et description du code

Les fichiers API_Gene_instance.h, API_Gene_instance.c, API_Grille.h, API_Grille.c sont fournis dans le projet.

Liste_case.h

```
/* Liste simplement chainée de case repérée par les coordonnées entières i,j */
typedef struct elnt_liste{
    int i,j;
    struct elnt_liste *suiv;
} Elnt_liste;
typedef Elnt_liste *ListeCase;
```

Liste_case.c

Cette bibliothèque permet de manipuler une structure de type Elnt_liste :

```
~   /* Initialise une liste vide */
    void init_liste(ListeCase *L);
~   /* Ajoute un element en tete de liste */
    int ajoute_en_tete(ListeCase *L, int i, int j);
~   /* teste si une liste est vide */
    int test_liste_vide(ListeCase *L);
~   /* Supprime l'element de tete et retourne les valeurs en tete */
    /* Attention: il faut que la liste soit non vide */
    void enleve_en_tete(ListeCase *L, int *i, int *j);
~   /* Detruit tous les elements de la liste */
    void detruit_liste(ListeCase *L);
~   /* Affiche les elements de la liste */
    void afficheListeCase(ListeCase *L);
```

Ce fichier inclut Liste_case.h.

Biblio_S_Zsg.h

```
typedef struct {
    int dim;            /* dimension de la grille */
    int nbcl;           /* nombre de couleurs */
    ListeCase Lzsg;     /* liste de cases de la zone Zsg */
    ListeCase *B;       /* tableau de listes de cases de la bordure reparties par couleurs, de taille nbcl */
                        /* chaque case B[c] est une liste contenant les cases de couleur c de la bordure */
    int **App;          /* tableau n x n a double entrée des appartenances */
} S_Zsg;
```

Ce fichier inclut Liste_case.h.

Biblio_S_Zsg.c

Cette bibliothèque permet de manipuler une structure de type S_Zsg :

```
~   /* Initialise une structure _Zsg */
    void init_Zsg(S_Zsg *Z, int dim, int nbcl);
~   /* Ajoute une case dans la liste Lzsg */
    int ajoute_Zsg(ListeCase *Lzsg, int i, int j);
~   /* Ajoute une case dans la bordure d'une couleur cl donnée ; */
    int ajoute_Bordure(ListeCase *B, int i, int j);
~   /* Renvoie vrai si une case est dans LZsg */
```

	int appartient_Zsg(S_Zsg *Z, int i, int j);
	~ /* Renvoie vrai si une case est dans la bordure de couleur cl donnée */
	int appartient_Bordure(S_Zsg *Z, int cl, int i, int j);
	~ /* Desalloue une structure S_Zsg */
	void libere_Zsg(S_Zsg *Z);
	
	Ce fichier inclut Biblio_S_Zsg.h.

Graphe.h	#define INFINITO 2140000000 /* Constante pour l'initialisation de la distance sommet-racine (EXO6) */
	
	/* Element d'une liste chainee de pointeurs sur Sommets */
	typedef struct cellule_som {
	Sommet * sommet; /* Pointeur sur sommet */
	struct cellule_som * suiv; /* Pointeur vers l'élément suivant */
	} Cellule_som;
	
	typedef struct sommet Sommet;
	struct sommet {
	int num; /* Numéro du sommet (sert uniquement a l'affichage) */
	int cl; /* Couleur d'origine du sommet-zone */
	ListeCase cases; /* Listes des cases du sommet-zone */
	int nbcase_som; /* Nombre de cases de cette liste */
	Cellule_som * sommet_adj; /* Liste des arêtes pointeurs sur les sommets adjacents */
	int marque; // pour EXO5 - Statut du sommet : 0 si dans Zsg, 1 si dans Bordure Zsg, 2 si non visité */
	int distance; // pour EXO6 - nombre d'arêtes reliant ce sommet a la racine */
	Sommet *pere; // pour EXO6 - père du sommet dans l'arborescence du parcours en largeur */
	};
	
	typedef struct graphe_zone {
	int nbsom; /* Nombre de sommets dans le graphe */
	int dimM; /* Dimension de la matrice M représentée et de la matrice mat */
	Cellule_som *som; /* Liste chainée des sommets du graphe */
	Sommet ***mat; /* Matrice de pointeurs sur les sommets indiquant a quel sommet
	} Graphe_zone; appartient une case (i,j) de la grille */

Entete_Fonctions.h	Ce fichier contient les prototypes des fonctions des exercices 1, 2, 3, 5, 6, 7. Ce fichier inclut API_Grille.h, Liste_case.h, Biblio_S_Zsg.h, Graphe.h.

Fonctions_exo1	La première méthode pour déterminer la zone supérieure gauche (Zsg) est la récursivité.
	
	La fonction *trouve_zone_rec* permet d'ajouter à la liste chainée en paramètre les cases de la zone qui contient la case i,j. Cette fonction prend en paramètre : la liste chainée, la matrice, la dimension de la matrice, les coordonnées de la case i, j et les nombre de cases qui se trouvent dans la zone ; la fonction met à jour ce nombre de cases et leur attribue la valeur -1 dans la matrice correspondante.
	void trouve_zone_rec(int **M, int nbcase,int i, int j, int *taille, ListeCase *L);
	
	La fonction *sequence_aleatoire_rec* utilise la fonction *trouve_zone_rec* pour déterminer la Zsg.
	Le choix de la couleur pour le nouveau tour de jeu est aléatoire, ainsi on affecte cette couleur à toutes les cases de la liste chainée. On effectue cette manipulation jusqu'au moment où la taille de la zone supérieure gauche recouvre toutes les cases. L'algorithme retourne le nombre de tours de jeu effectués. La grille est mise à jour et affichée et on retourne le nombre de fois qu'on a colorié les cases.
	int sequence_aleatoire_rec(int **M, Grille *G, int dim, int nbcl, int aff);
	
	Ce fichier inclut Entete_Fonctions.h.

Fonctions_exo2	Lors de la méthode de l'exercice 1 on a remarqué que la récursivité est limitée en nombre d'exécutions (erreur de segmentation), donc cette méthode sera non récursive.
	
	La fonction récursive *trouve_zone_rec* est transformée dans une nouvelle fonction non récursive *trouve_zone_imp.* Cette fonction a les mêmes paramètres que la précédente. Elle utilise une pile et les fonctions *ajouter_en_tete* et *enleve_en_tete* . On ajoute la case i,j à la pile, on vérifie si cette case est de la même couleur que la Zsg : si tel est le cas, on ajoutera cette case à la liste chaînée L et on traitera les cases adjacentes de la même couleur. Finalement, on éliminera de la pile l'élément qu'on vient de traiter.
	La fonction retourne dans L la liste des cases de même couleur que la case i,j et met -1 dans ces cases.
	void trouve_zone_imp(int **M, int nbcase,int i, int j, int *taille, ListeCase *L);
	
	D'une manière analogue à celui de l'exercice 1, cet algorithme tire au sort une couleur et utilise la fonction trouve_zone_imp pour déterminer la Zsg.
	L'algorithme retourne le nombre de tours de jeu effectués.
	int sequence_aleatoire_imp(int **M, Grille *G, int dim, int nbcl, int aff);
	
	Ce fichier inclut Entete_Fonctions.h.

Fonctions_exo3	La fonction a comme objectif de mettre à jour la liste Lzsg et le tableau B de cases de la bordure. Au début, on initialise une pile P ne contenant que la case (0,0). Tant que la pile n'est pas vide, on examine le dernier élément ajouté à P : si la case examinée est de couleur cl et si elle n'appartient pas déjà à Lzsg on ajoute la case i,j à la liste Lzsg et on collecte les cases adjacentes à la Zsg pas encore traitées ; sinon, si elle n'appartient pas déjà à B[cl], on ajoute la case i,j à la liste B[cl] et on élimine de la pile l'élément qu'on vient de traiter.
	La fonction retourne le nombre de cases qui a été ajouté à Lzsg.
	int agrandit_Zsg(int **M, S_Zsg *Z, int cl, int k, int l);
	
	Algorithme tirant au sort une couleur: il utilise la fonction agrandit_Zsg pour déterminer la nouvelle bordure B et agrandir la Zsg. Tant qu'on n'a pas remplie la grille avec une seule couleur on choisit une nouvelle couleur différente de celle de la zone supérieure gauche et on affecte cette couleur à la Lzsg. À la fin, on redessine la grille, on libère la mémoire et on retourne le nombre de tours de jeu effectués.
	int sequence_aleatoire_rapide(int **M, Grille *G, int dim, int nbcl, int aff);
	
	Ce fichier inclut Entete_Fonctions.h.

Graphe.c	Fonction qui alloue, construit et affiche une structure Graphe_zone a partir des informations contenues dans la matrice M. Chaque sommet représentera une zone de cases adjacentes dans M caractérisées par la même couleur. Ce fichier inclut Entete_Fonctions.h.

Fonctions_exo5	Les fonctions de l'exercice 5 sont similaires a celles du 3, mais elles s'appliquent à une structure Graphe_zone.
	
	La fonction met à jour les champs Lzsg et B d'une S_Zsg lorsque l'ensemble des cases qui appartiennent à une même zone (sommet) de couleur cl (dans la bordure B[cl]) doit basculer dans Lzsg.
	On parcourt la liste B[cl] de la même couleur que la case passée en paramètre.
	On ajoute a la liste Lzsg toutes les cases contenues dans la liste B[cl] et on marque à 0 les sommets correspondants. Pour toute nouvelle case acquise dans Lzsg (et relatif sommet dans J->mat) :
	~ on explore la liste de sommets adjacents 'sommet_adj'. Pour chacun de ceux la :
	~ on explore la liste de cases 'cases' (et relatif sommet dans J->mat). Si le sommet n'a pas encore été visité, (marque != 0) alors on ajoute sa case a la bordure-graphe B[cl] de son couleur.
	On met à jour la marque à 1 pour tous les sommets ajoutés à la bordure-graphe.
	La fonction retourne le nombre de cases ajouté à Lzsg.
	int agrandit_BordureGraphe(Graphe_zone *J, S_Zsg *Z, int k, int l);
	
	L'algorithme colorie le graphe représentant la grille du jeu à partir de la stratégie suivante : à chaque tour de jeu, la nouvelle couleur choisie est celle la plus représentée dans la bordure-graphe. Pour l'implémentation de la strategie_maxBordure on a utilisé la structure existante S_Zsg. L'algorithme retourne le nombre de tours de jeu effectués.
	int strategie_maxBordure(Graphe_zone *J, Grille *G, int dim, int nbcl, int aff);
	Ce fichier inclut Entete_Fonctions.h.

Fonctions_exo6	L'algorithme calcule le parcours en largeur d'un graphe à partir d'un sommet racine. Pour chaque sommet du graphe, on stocke le champ distance (distance sommet-racine) et le père (sommet prédécesseur). En remontant l'arborescence a partir d'un sommet destination, en suivant le pointeur 'père' vers son prédécesseur, il est possible de reconstruire un plus court chemin entre les sommets racine et destination. void parcoursLargeur(Graphe_zone *J, Sommet *racine) ; L'algorithme utilise la fonction parcoursLargeur pour déterminer le plus court chemin entre la première (M[0][0]) et la dernière (M[dim-1][dim-1]) case de la grille de jeu. Ce parcours nous permet de trouver la séquence de couleurs minimale pour colorier la grille en diagonale jusqu'a la case en bas a droite. Stratégie : on colorie en priorité les zones (sommets) correspondantes au plus court chemin mentionné, ensuite on termine le jeu en suivant la stratégie strategie_maxBordure de l'exercice 5. L'algorithme retourne le nombre de tours de jeu effectués. int strategie_parcoursLargeur_MaxBordure(Graphe_zone *J, Grille *G, int dim, int nbcl, int aff) ; Ce fichier inclut Entete_Fonctions.h.
Fonctions_exo7	En partant de l'exercice 6, on a implémenté cette stratégie afin d'améliorer les performances du code en terme de taille de la séquence de couleurs . Cette fonction recalcule le plus court chemin dans le graphe et l'enregistre dans une liste fileSommets, en appliquant les stratégies suivantes: ~ stratégie 1 : plus court chemin a partir de M[0][0] vers M[dim/2][dim-1] ~ stratégie 2 : plus court chemin a partir de M[dim/2][dim-1] vers M[dim-1][dim/3] void recalcule_fileSommets(Graphe_zone *J, int nStr, int dim, Cellule_som **ListeCellS) ; L'algorithme utilise la fonction recalcule_fileSommets pour définir la séquence de couleurs a suivre pendant le déroulement du jeu. Les plus courts chemins sélectionnés permettent de colorier les sommets du graphe selon des axes transversales qui coupent l'instance et rejoint rapidement les coins de la grille, normalement les plus compliqués à inonder. Ensuite on termine le jeu en suivant la stratégie strategie_maxBordure de l'exercice 5. L'algorithme retourne le nombre de tours de jeu effectués. NB. Le code BounceStrategyBonus est commenté seulement dans les parties qui se différencient de l'exercice 6, pour faciliter la lecture. int BounceStrategyBonus(Graphe_zone *J, Grille *G, int dim, int nbcl, int aff) ; Pour des grilles de dimensions réduites cette méthode n'est pas profitable, mais, comme on le verra dans l'analyse statistique, elle permet de réduire la quantité d'essais nécessaires pour terminer le jeu, en proportion avec la dimension et le nombre de couleurs. Ce fichier inclut Entete_Fonctions.h.
Flood-It	Fonction main du programme.
Makefile	Compilation et édition des liens entre les fichiers du programme.

Description des Jeux de test

Exécution du programme : ./Flood-It <dimension> <nb_couleurs> <niveau> <graine> <exo : 0-1-2-4-5-6-7> <affichage : 0-1>
NB. 0-1-2 correspondent aux exercices 1-2-3 et 4-5-6-7 correspondent aux exercices 4-5-6-7. Il n'y a pas de lancement <exo : 3>.

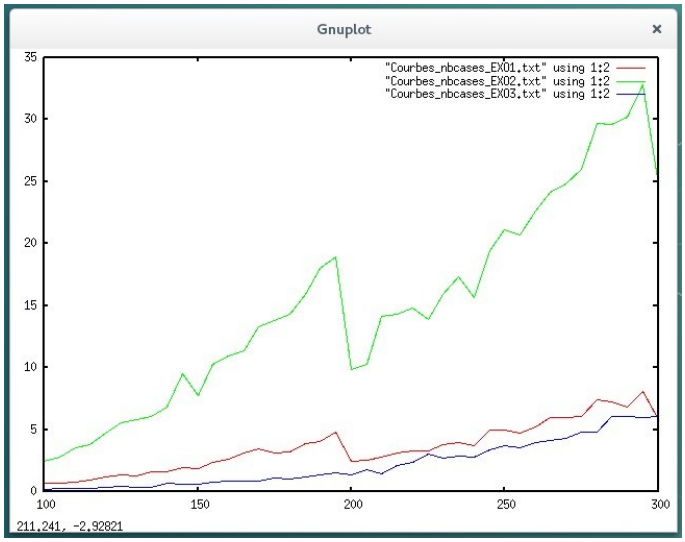
Dans les exercices 5 et 6 on a laissé en commentaire des parties de code qui montrent en détail le fonctionnement du programme. Pour les utiliser en toute simplicité, il faut juste déplacer le symbole « */ » situé en bas à gauche du paragraphe à la fin de la ligne pointillée qui contient l'en-tête « Jeu de test ».

Analyse des performances

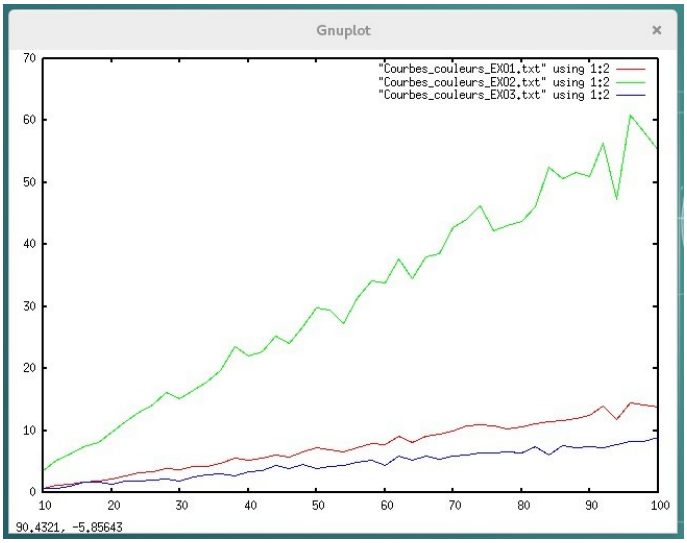
La comparaison de l'ensemble des versions de l'exercice montre une nette amélioration des performances en terme d'essais nécessaires dans les trois dernières versions par rapport aux trois premières. Dans les exercices 1,2 et 3 on conclut le jeu à travers un tirage aléatoire des couleurs ; alors que dans les exercices 5, 6 et 7 on met à point des stratégies qui visent à obtenir la meilleure séquence de couleurs apte à la résolution du jeu.

Analyse des exercices 1-2-3

Les graphiques obtenus montrent les performances des 3 versions de programmes (EXO1, EXO2, EXO3) en fonction de la dimension (Courbes_nbcases) et du nombre de couleurs (Courbes_couleurs).
À l'augmenter de la dimension, les temps d'exécution augmentent proportionnellement.
À l'augmenter du nombre de couleurs, les temps d'exécution augmentent proportionnellement.
Pour les 2 représentations, la version de l'exercice 2 (impérative) résulte largement la plus lente, ensuite on a la version de l'exercice 1 (récursive) ; la version de l'exercice 3 (structure acyclique) résulte la plus rapide.
Conclusion : la version de l'exercice 3 assure les meilleurs prestations exprimées en temps d'exécution par rapport au nombre de cases et nombre de couleurs.



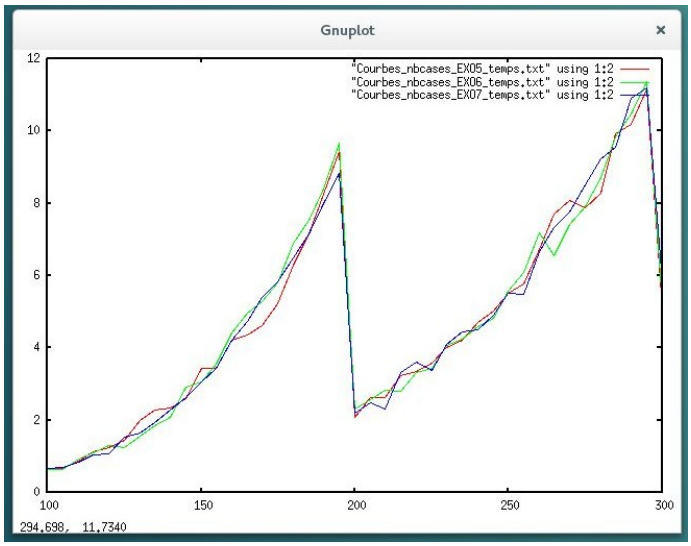
Comparaison en terme de temps d'execution et de nombre des cases des exercices 1 (rouge), 2 (vert) et 3 (bleu)



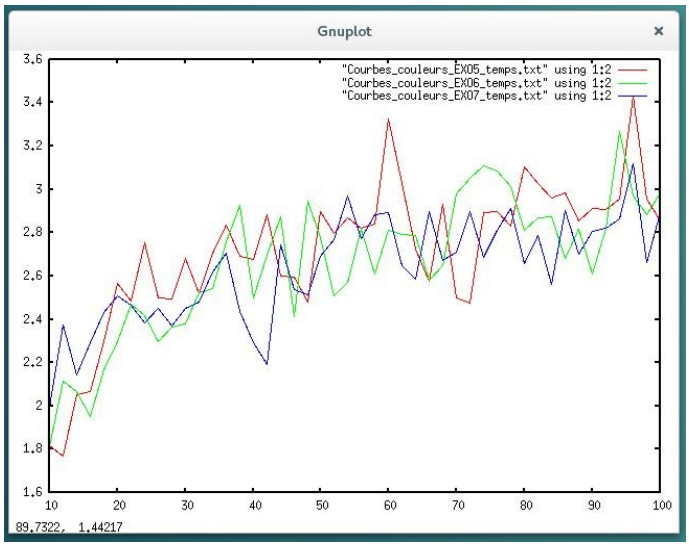
Comparaison en terme de temps d'execution et de nombre des couleurs des exercices 1 (rouge), 2 (vert) et 3 (bleu)

Analyse des exercices 5-6-7

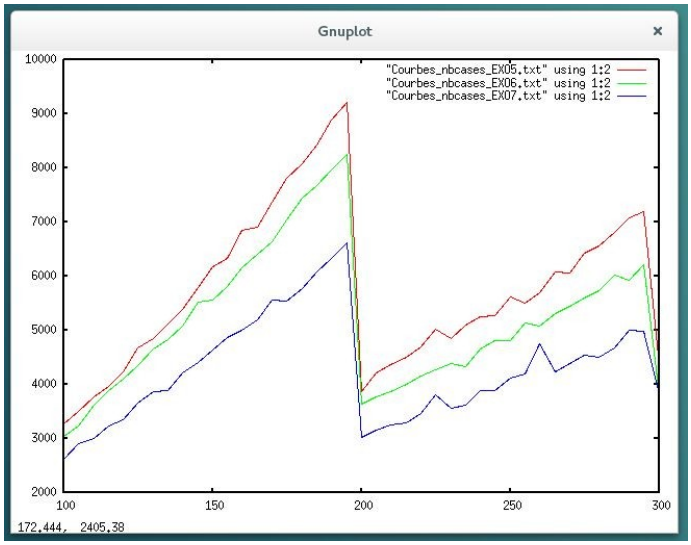
Les graphiques obtenus montrent les performances des 3 versions de programmes (EXO5, EXO6, EXO7) en fonction de la dimension (Courbes_nbcases) et du nombre de couleurs (Courbes_couleurs).
Les deux premiers graphiques ci-dessous montrent que, au niveau des temps d'exécution, les trois versions de l'exercice sont à peu près équivalents. En effet, même si les exercices 5, 6 et 7 résolvent la grille plus rapidement avec moins de couleurs, la création du graphe demande un temps initial supplémentaire.On a donc décidé de souligner les différences de prestation à travers la comparaison des nombres d'essais nécessaires pour remplir la grille.
Dans les deux derniers graphiques on pourra donc remarquer que, soit en fonction de la dimension que du nombre des couleurs, la version de l'exercice 5 demande le plus d'essais, alors que la version de l'exercice 6, en coupant en deux l'instance, en demande moins. En partant de l'exercice 6, on a implémenté la même stratégie en coupant l'instance deux fois : une axe transversale arrive rapidement au point M[dim/2][dim-1], ensuite une deuxième axe partant de ce point va vers M[dim-1][dim/3]. Selon les analyses effectuées, la version de l'exercice 7 résulte en général la plus performante. Pour des grilles de dimensions réduites cette méthode n'est pas profitable, mais dès qu'on augmente la dimension et le nombre de couleurs elle permet de réduire proportionnellement la quantité d'essais nécessaires pour terminer le jeu. Conclusion : la version de l'exercice 7 assure les meilleurs prestation de l'ensemble des exercices du projet exprimées en nombres d'essais par rapport au nombre de cases et au nombre de couleurs.



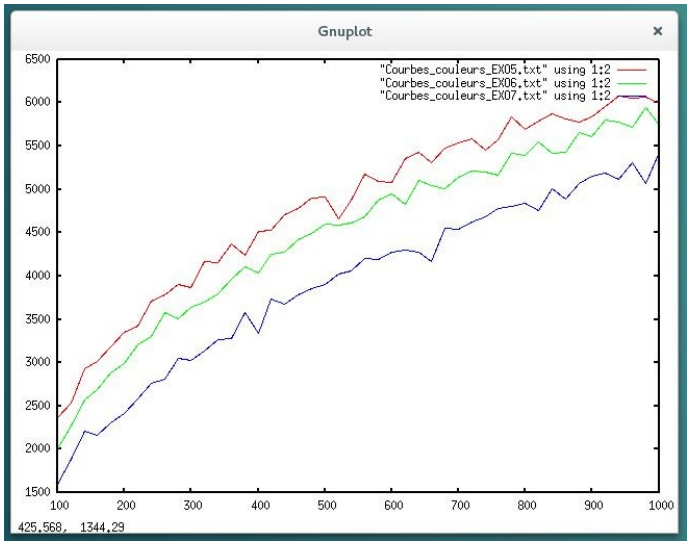
Comparaison en terme de temps d'execution et de nombre des cases des exercices 5 (rouge), 6 (vert) et 7 (bleu)



Comparaison en terme de temps d'execution et de nombre des couleurs des exercices 5 (rouge), 6 (vert) et 7 (bleu)



Comparaison en terme de nombre d'essais et de nombre des cases des exercices 5 (rouge), 6 (vert) et 7 (bleu)



Comparaison en terme de nombre d'essais et de nombre des couleurs des exercices 5 (rouge), 6 (vert) et 7 (bleu)

Libération de la mémoire allouée

On a remarqué une fuite de mémoire due à la partie du code fournie, visible dans l'exercice 0 en choisissant le lancement avec ou sans affichage (in use at exit : 44.872 bytes in 504 blocks).
On signale en outre les fuites de mémoire suivantes :

- Exercice 3 (<exo : 2>) : in use at exit : 8 bytes in 1 block
- Exercices 4-5-6-7 (<exo : 4-5-6-7>) : il reste des résidus de mémoire non libérée, en relation avec la structure Sommet.