Nicolas CUADROS, Sébastien BERLIOUX, Simon BESSENAY
Ximeng ZHANG, Antoine MARJAULT

HashCode 2018

Nicolas CUADROS, Sébastien BERLIOUX, Simon BESSENAY
Ximeng ZHANG, Antoine MARJAULT

I. Introduction

Objectif

Avant de démarrer le compte-rendu de ce projet il est important de rappeler brièvement les objectifs initiaux.
Le projet s'intéresse à une problématique mondiale actuelle : la forte croissance démographique des grandes villes.
La problématique globale consiste donc à trouver une répartition optimale de bâtiments résidentiels et utilitaires dans un espace vierge donné. Par *optimale* on entend maximiser le nombre d'habitants à proximité de bâtiments utilitaires (boulangerie, poste, …).
Pour cela nous avons accès à 6 cartes différentes composées de bâtiments différents et comportant donc des problématiques assez différentes.

L'objectif final est donc - à partir d'un fichier d'entrée - de produire une carte remplie des différents bâtiments listés dans ce dernier et placés de façon optimale.
Pour cela nous avons dû procéder en 2 étapes : le développement d'un *arbitre* qui sert à tester la validité et le score de la solution produite et le développement du programme principale permettant de résoudre une carte.

Organisation de l'équipe

Le projet s'est divisé en trois grandes phases : la conception de l'architecture du projet, le développement de l'*arbitre*, le développement des classes (le "squelette" du programme) et enfin le développement d'un algorithme de résolution.
La partie conception a été réalisée par Nicolas CUADROS et Simon BESSENAY, avec comme finalité un diagramme de classe qui a été transmis à l'équipe afin de tous se mettre d'accord sur la marche à suivre et ainsi se répartir le développement des différentes classes.
Au cours du projet l'architecture n'a quasiment pas évoluée en terme de structure mais beaucoup de méthodes et d'attributs ont étés ajoutés selon les besoins, preuve que la structure avait été bien réfléchie.
L'arbitre consiste, comme expliqué précédemment, à vérifier que la solution est valide (*i.e* les bâtiments ne se chevauchent pas, les bâtiments sont entiers, etc) et à donner le score obtenu par une solution.
La partie développement des classes de base a été répartit assez équitablement entre les membres du groupe, chacun ayant au moins 2 classes à coder. Cette répartition a permis de développer chacun de son côté sans gêner les autres (même si certaines classes nécessitent l'existence de d'autres cela s'est plutôt bien déroulé).
Vers le milieu du développement du squelette du programme nous avons commencé à réfléchir à différentes solutions (que nous détaillerons dans la partie suivante) pour au final en retenir une. Au départ nous comptions trouver 2 ou 3 solutions et les développer en parallèle sur des branches du projet et garder la meilleure mais une solution semblait être la plus efficace et par manque de temps dû aux autres projets et rendus en cours nous avons décidé de se concentrer uniquement sur celle-ci.

II. Stratégie

Afin de simplifier la compréhension des explications qui suivent, nous allons décrire brièvement le fonctionnement global du programme. La classe *Project* est centrale dans l'architecture du projet : on crée une instance globale de celle-ci au lancement du programme. Cette instance contient toutes les informations contenues dans le fichier d'entrée ainsi que la carte contenant la solution intermédiaire/finale.

Parsing et synthèse des I/O

La première étape consiste à lire le contenu du fichier d'entrée (au format .in) et de charger les données dans l'instance globale de la classe *Project.* Cette instance contient principalement des vectors permettant de stocker des objets *Buildings* pouvant être soit des *Utility* soit des *Residential*, elle contient aussi une *City* qui représente en fait une carte et qui contiendra à la fin de l'exécution la solution finale. Au lancement de l'exécutable, le chargement du fichier d'entrée se fait au moyen d'une classe *FileLoader* qui va lire le fichier d'entrée grâce à sa méthode *loadProject* et créer les différents *Residential* et *Utility* (héritants de la classe *Building*) contenus dans celui-ci puis les ajouter au *Project*. Ensuite on va créer une *City* vide de la même taille que celle indiquée dans le fichier. Enfin, les différentes informations seront elles aussi chargées dans le projet (*maxWalkingDistance, …).*
A la fin de l'exécution du programme, lorsque la solution a été calculée on a alors l'attribut *City* du projet qui la contient. Un simple appel à la méthode *toSolution* de la classe *City* permet d'écrire la solution dans le fichier de sortie précisé en paramètre sous la forme indiquée dans l'énoncé du HashCode.

Méthode de résolution

Plusieurs idées de méthodes de résolution ont été évoquées au départ, certaines paraissaient efficaces mais demandaient trop de puissance de calcul et d'autres étaient très rapides mais possédaient des inconvénients dont le principal était la diminution du score.
La première idée était de résoudre le problème par force brute en plaçant les bâtiments un par un côte à côte en testant à chaque itération tous les bâtiments possibles à chaque position voisine des bâtiments déjà placés. Après quelques calculs il était clair que la résolution du problème avec cette méthode sur une carte de taille 1000 x 1000 impliquait beaucoup trop de calculs.
Nous avons donc eu l'idée de segmenter la carte en plusieurs sous-cartes qui seraient calculées indépendamment puis assemblées à la fin du programme.

Nicolas CUADROS, Sébastien BERLIOUX, Simon BESSENAY
Ximeng ZHANG, Antoine MARJAULT

Résolution d'une sous-carte

Pour encore optimiser le temps de calcul, nous avons décidé de calculer les sous-cartes de façon plus intelligente :
La méthode consiste à d'abord remplir une carte en plaçant des bâtiments de droite à gauche et de bas en haut, de droite à gauche et de haut en bas pour finir par remplir les derniers trous en plaçant des bâtiments aléatoirement sur la carte (en vérifiant qu'il n'y a aucun chevauchements). Le fait de balayer la carte dans différents sens permet une répartition plus variée des bâtiments.
La deuxième étape consiste à calculer les ensembles vides connexes de la carte et de repérer ceux dont la taille est supérieure à celle du bâtiment le plus petit plaçable et à remplir ces zones. Le fait de vérifier la taille permet d'être sûr qu'au moins un bâtiment (le plus petit) pourra être placé dans la zone vide.
La dernière étape est en fait un post-traitement qui applique le même principe qu'à l'étape précédente mais en prenant cette fois-ci les bâtiments donnant le moins bon ratio score/densité et en les remplaçant par d'autres bâtiments.
Cette méthode de placement donne de bons résultats et possède l'avantage d'être très rapide sur des sous-cartes de petite taille (une 100x100 se remplit en moins d'une seconde).

L'assemblage des sous-cartes est effectué une fois leur remplissage terminé. Il consiste à placer les sous-cartes une par une côte à côte de gauche à droite et de haut en bas en plaçant à chaque fois celle qui donne le meilleur score lorsqu'elle est placée. Cette technique implique un temps d'assemblage assez long puisque pour chaque case (de la taille de la sous-carte) de la carte finale il faut tester une par une les sous-cartes calculées précédemment et ne laisser que la meilleure, impliquant ainsi des créations/destructions coûteuses d'objets *City*. Néanmoins on observe un score supérieur à un assemblage classique (consistant à mettre côte à côte les sous-cartes dans un ordre quelconque).

Le premier avantage de notre solution est qu'elle est modulable : en réglant certains paramètres tels que la taille d'une sous-carte ou le nombre de sous-cartes à calculer on peut adapter le rapport temps de calcul / score. Si on veut un résultat rapide avec un score un peu moins bon on peut par exemple séparer une carte de taille 1000x1000 en sous-cartes de 100x100 (temps de calcul compris entre 5 et 15 minutes selon les cartes) alors que si on veut une solution avec un score maximal mais dont le calcul sera plus lent on peut calculer une seule sous carte de la taille initiale de 1000x1000. En pratique on obtient même de meilleurs score en séparant la carte en sous-cartes de façon générale.
Le deuxième avantage est qu'il est possible de répartir le calcul des sous-cartes sur différents threads puisqu'ils sont indépendants. On peut alors diviser grandement le temps de calcul.

Nous avons pensé à améliorer notre solution en réalisant un post-traitement après l'assemblage des sous-cartes qui permettrait d'éviter l'*effet de bord* dû au fait que les bâtiments placés sur les bords de deux sous-cartes ne sont pas forcément "compatibles". Cependant la tâche semblait très lourde en calcul nous avons donc décidé de laisser le programme tel quel.

Nicolas CUADROS, Sébastien BERLIOUX, Simon BESSENAY
Ximeng ZHANG, Antoine MARJAULT

Optimisation

a. Optimisation de l'algorithme

Comme expliqué plus haut nous avons utilisé des poids assimilés à des probabilités pour le choix des bâtiments à placer. Ces poids ont été optimisés "à la main" en faisant des tests sur plusieurs dizaines de sous-cartes en comparant le score moyen en fonction des 4 poids. Afin d'optimiser encore plus le score total (de toutes les cartes) les poids peuvent être ajustés pour chaque carte. Ce processus pourrait être automatisé si on devait traiter un grand nombre de carte au moyen d'algorithmes d'apprentissage mais ce n'est pas l'objectif ici c'est pourquoi nous nous sommes contentés d'optimiser les poids à la main.
La deuxième optimisation qui a été faite est l'utilisation de thread qui divisent grandement le temps de calcul puisque ces derniers sont effectués en parallèle.

b. Optimisation du code

Vers la fin du projet nous avons pu optimiser la qualité du code notamment au niveau de la gestion de mémoire et du temps de calcul des fonctions les plus lourdes.
Valgrind a permis de retirer les fuites de mémoires présentes dans certaines parties du code et *Gprof* nous a permis de déterminer que la fonction de placement de building était la plus gourmande, cependant malgré plusieurs tentatives nous n'avons pas réussi à l'optimiser.

III. Conclusion

En résumé, nous avons produit un arbitre fonctionnel et un programme qui donne des solutions pour les différents fichiers d'entrés avec de bons scores.
Nous sommes très satisfait du programme que nous avons conçu, d'autant plus que son architecture permet de modifier les différents algorithmes de placement, d'assemblage et de sélection très facilement permettant ainsi une évolution durable dans le temps.
Le code est propre, facilement maintenable et surtout il a été bien optimisé pour éviter les fuites de mémoires par exemple. La force de notre programme est sa rapidité : les cartes mettent entre 5 et 15 minutes pour être résolues.

Pour finir, ce projet nous a permis une fois de plus d'expérimenter le travail en groupe nous rappelant à quel point il est important de communiquer et de se partager le travail.