Projet NPM3D

OctoMap: A Probabilistic, Flexible, and Compact3D Map Representation for Robotic Systems

Lucas Pluvinage

Avril 2020

1 Introduction

L'article présente *OctoMap*, une structure de données probabiliste permettant d'intégrer les résultats de scans lasers dans une carte 3D à résolution variable. Cette structure est un octree dont l'utilisation mémoire a été optimisée et permettant de stocker sous forme probabiliste les états des cellules. L'encodage des données scanner des probabilités (au lieu d'une information binaire: case pleine/vide) permet une intégration plus fine des données scanner car elle prend en compte le bruit intrinsèque au monde réel. Aussi l'article se concentre sur des questions d'efficacité algorithmique et de représentation flexible de données probabilistes. L'implémentation est open-source, disponible sur Github sous licenses New BSD (octomap) et GPL (octovis, le logiciel de visualisation associé): *github.com/octomap/octomap*.

Une représentation mémoire efficace est présentée, ainsi que le format de stockage de fichier. Il y a deux format de fichiers: un format ultra-compact stockant seulement si chaque cellule est pleine, vide ou inconnue, et un format sans perte stockant l'ensemble des informations probabilistes sous forme de *log-odds* (nombres flottants 32 bits).

Mon ambition pour le projet fut de questionner cette dichotomie en permettant un choix variable de la précision. Le but étant de permettre un compromis entre la précision des valeurs et l'utilisation de la mémoire. C'est le stockage sérialisé (sous forme de fichier) qui fut étudié, car c'est là où la dichotomie intervient. En effet en mémoire les données sont stockées de la même façon dans les noeuds, et la possibilité d'optimisation est moindre car le coût principal est celui des pointeurs et non celui des valeurs. L'enjeu est donc d'étudier la taille des fichiers générés par octomap et de chercher à optimiser cette taille, tout en gardant une complexité temporelle comparable.

Les données liées au projet, ainsi que l'implémentation réalisée, sont disponible sur Github:
github.com/TheLortex/octomap-project

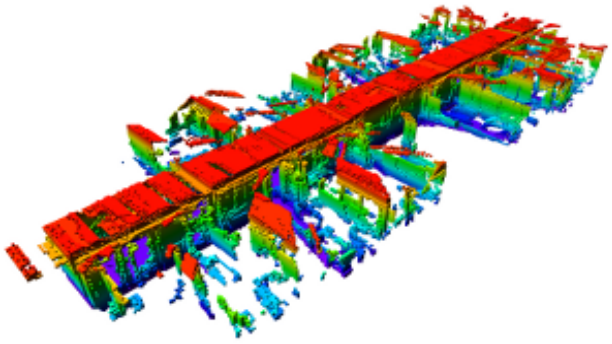

Figure 1: Scan d'un couloir de l'université de Freiburg
(image prise de *http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/*)

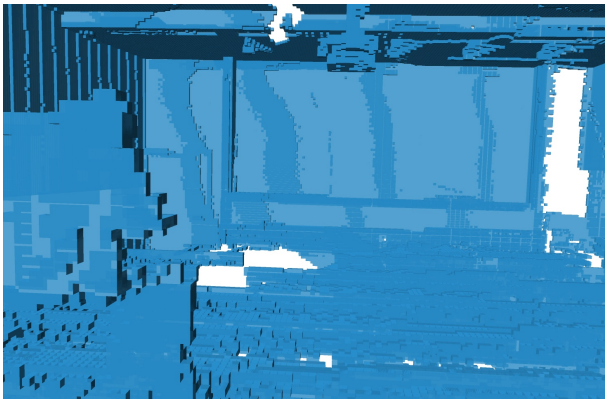

Figure 2: Scan d'une salle de l'université de Würzburg

2 Reproduction des résultats, analyses

Pour mesurer l'utilisation de mémoire des différents formats de stockage, les jeux de données suivants furent utilisés:

- Freiburg 079 corridor[1]: 45x20x3m, 117MB
- Freiburg campus: 300x170x30m, 445MB
- Würzburg university lecture hall[2]: 435MB. Le fichier de scan fut converti en fichier `.log` par un script que j'ai développé avant d'être transformé par `log2graph` présent dans le projet octomap.

Pour chaque jeu de donnée, le programme `graph2tree` est utilisé générant trois fichiers.

Le premier fichier est en `.ot` et contient l'ensemble des probabilités sous forme de *log-odds*, chaque valeur prenant alors 4 octets. Le deuxième fichier est en `_ml.ot` et stocke les maximum de vraisemblance pour chaque valeur, cela a pour effet réduire la taille de l'arbre car beaucoup de cellules se retrouvent avec la même valeur. Enfin le format `.bt` est le format le plus optimisé, stockant deux bits d'information pour chaque feuille. Ce dernier se base sur le format en maximum de vraisemblance et extrait 2 bits d'information de chaque valeur, les 4 possibilités étant: feuille pleine, vide, inconnue, ou noeud ayant des enfants.

3 Extension et implémentation

Il est facile de remarquer qu'entre deux bits et 4 octets par valeur, il y a un saut énorme dans la quantité d'information transmise. On peut se demander s'il est intéressant de permettre des précisions intermédiaires pour le stockage de l'information.

J'ai donc exploré les possibilités offertes par le stockage sur 1 à 32 bits de nombres à virgule fixe. L'idée est simplement de discrétiser l'espace des valeurs entre 0 et 1 de façon uniforme avec un nombre de valeur fixé en fonction de la précision voulue. Chaque probabilité est approximée par $k/(2^n - 1)$ avec n le nombre de bits de précision donné. Par exemple, 1 à 4 octets correspondent respectivement à 256, 65536, 16 millions et 4 milliards de valeurs possibles. Pour représenter sur n bits les valeurs de probabilités des feuilles, il suffit de multiplier la valeur par la valeur maximale d'un entier représentable sur ces n bits et d'arrondir à l'entier le plus proche. Le décodage s'opère symétriquement par division.

Cette compression s'accompagne évidemment d'une perte de précision dans les valeurs encodées. On observera cependant que l'erreur devient suffisamment faible à partir de deux octets de précision:

3.1 Implémentation du stockage optimisé à précision contrôlable

Le projet octomap étant open-source, il me fut aisé d'obtenir le code et de le compiler. J'ai pu ainsi comprendre l'architecture du projet et en déduire là où les modifications devraient être apportées. Pour simplifier l'implémentation, seuls les encodages en octets furent supportés: c'est à dire 8, 16, 24 et 32 bits. En effet pour les précisions intermédiaires la complexité du code grandit, car alors qu'il est facile

[1] OctoMap 3D scan dataset by University of Freiburg. http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/
[2] by Johannes Schauer from the University of Würzburg, Germany. http://kos.informatik.uni-osnabrueck.de/3Dscans/

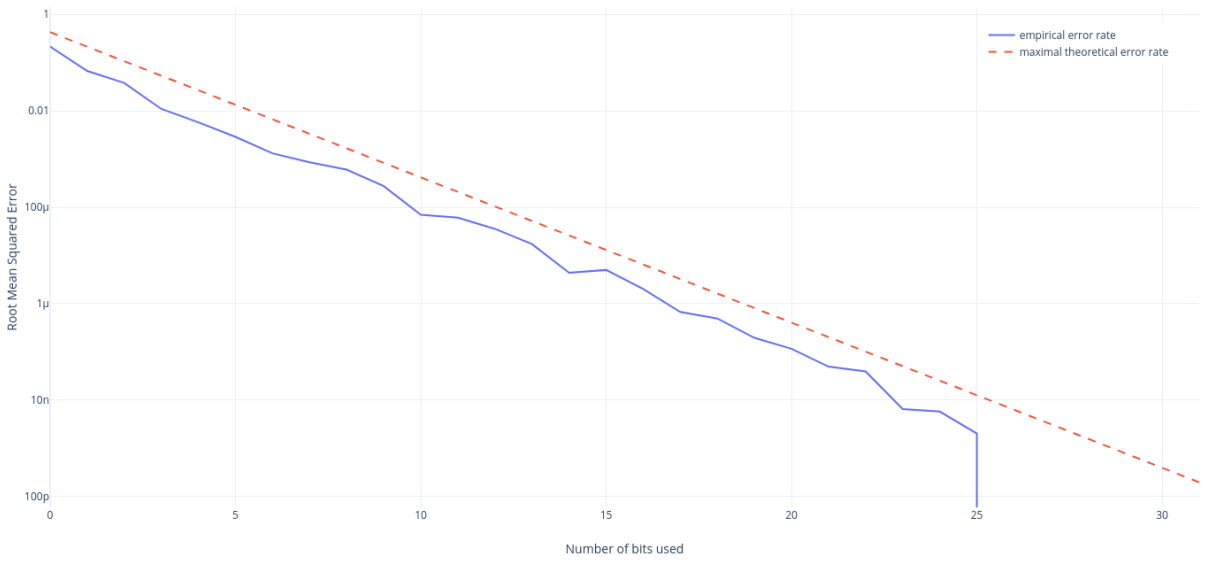

Figure 3: Erreur moyenne en fonction du nombre de bits de précision. Échelle logarithmique. Note: à partir de 26 bits de précision l'erreur moyenne devient nulle, la précision des flottants 32 bits devenant le facteur limitant.

d'écrire un fichier octet par octet, l'écriture bit par bit nécessite d'agréger les bits voisins en octets avant de les écrire. Dans octomap les fonctions `read/writeBinary` se chargent de cela mais elles ne sont pas génériques et ne gèrent donc pas l'écriture d'un nombre de bits arbitraire par valeur.

Il a donc d'abord fallu modifier les signatures des fonctions de lecture/écriture (`readData` et `writeData`, entre autres) pour ajouter un paramètre `encoding` donnant le nombre d'octets de précision désiré.

Ensuite les en-têtes des fichiers `.ot` furent modifiées pour enregistrer la précision utilisée. Ce fut réalisé en modifiant les fonctions `readHeader` et `write` de la classe `AbstractOctree`.

Enfin `OcTreeNode.cpp` s'est vu ajouter les fonctions `writeData` et `readData` pour remplacer les fonctions héritées de `OcTreeDataNode.hxx`. C'est ces fonctions qui s'occupent de l'encodage/décodage. Ainsi cette partie reste complètement transparente pour le reste du projet. Les probabilités étant stockées au format *log-odds*, elles sont d'abord converties en nombres entre zéro et un avant de passer par l'algorithme d'encodage. En fonction du paramètre `encoding` 1 à 4 octets sont générés par valeur.

Le *diff* complet des modifications est disponible dans ce *commit* sur Github: *Support fixed-precision probabilities for bt storage.*

Par ailleurs un nouveau fichier `octree_precision.cpp` fut créé[3], il génère un exécutable du même nom permettant d'évaluer la perte de précision en fonction du nombre de bits utilisés. C'est ce que j'ai utilisé pour générer la figure 3.

4 Résultats

	2 bits	1 octet	2 octets	3 octets	4 octets
Freiburg campus	2.1M/1.3M	20M/5M	30M/5.7M	40M/6M	50M/6.4M
Freiburg corridor	149K/91K	1.2M/334K	1.7M/390K	2.3M/413K	2.9M/442K
Lecture hall	4.2M/1.9M	36M/4.1M	54M/4.6M	72M/4.0M	90M/5.2M

Figure 4: Taille des fichiers générés en fonction de la précision utilisée. *non compressé/compressé par gzip*

Comme on peut le voir dans le tableau 4, le choix de la précision de l'encodage a un effet non-négligeable sur la taille des fichiers obtenus, même une fois compressés. Par exemple en utilisant seulement 16 bits d'information au lieu de 32, on obtiens une compression d'environs 40% sans vraiment

[3]Commit *Binary to measure the precision loss on a tree*

perdre beaucoup de précision. De plus l'avantage de cette méthode est qu'elle permet de choisir plus précisément le compromis taille-précision voulu.

Temps d'exécution Cette méthode ajoute un peu de complexité dans le code de lecture/écriture de fichiers. Pour estimer le coût de l'encodage, j'ai utilisé `convert_octree` sur le jeu de données *lecture hall*, en lisant depuis un fichier standard et en écrivant dans un nouveau fichier avec l'encodage spécifié.

	Pas d'encodage	1 octet	2 octets	3 octets	4 octets
Temps	2.30s	2.55s	2.85s	3.12s	3.47s

Ainsi le coût total n'a pas de valeur en soi, c'est plutôt l'évolution du temps pris qui est intéressante. On remarque notamment que même si la méthode ajoute un certain coût temporel, ce coût est négligeable par rapport au temps de lecture et conversion des données scanner (de l'ordre de la minute).

5 Conclusion

En conclusion, on peut voir qu'il existe des moyens d'améliorer OctoMap de façon a permettre encore plus de flexibilité dans la représentation des données. Ce stockage a précision variable permet ainsi une réduire de l'espace disque utilisé, ce qui est non négligeable lorsque des grands jeux de données sont utilisés. Les modifications sont directement intégrables dans OctoMap même si une étape d'ingénierie logicielle supplémentaire serait nécessaire pour que le code soit en cohérence avec l'API de la bibliothèque.

De plus il est possible d'étendre cette flexibilité, en permettant de donner la précision en bits, et non en octets. Cela demande un effort de programmation plus fort et a un coût technique sur la lisibilité du code. Cependant même cela reste possible et envisageable, il n'est peut-être pas nécessaire d'avoir une telle granularité dans la précision.

Ce projet fut enrichissant dans la mesure où le comportement interne d'OctoMap fut exploré, et a mené à une implémentation fructueuse. Ainsi j'ai pu découvrir les enjeux du stockage et de la représentation des informations dans un cadre où les données se comptent par millions et où chaque optimisation compte.