Complexité et Calculabilité 2018/19

Ceci est la deuxième partie du sujet de complexité et calculabilité. Il contient une rapide explication du contenu de l'archive de base de code, ainsi qu'une explication du travail à réaliser.

1 Contenu de l'archive

L'archive qui vous est fourni contient :

- Un fichier `README` détaillant comment construire les différents exécutables possible, ainsi qu'un rappel du travail à réaliser.
- Un fichier `Makefile`.
- Un fichier `doxygen.config` construisant la documentation du projet.
- Un répertoire `parser` contenant le code du parseur, qu'il vous est inutile de regarder.
- Un répertoire `include` contenant les fichiers d'entête utiles à votre programme.
- Un répertoire `src` contenant `Graph.c` et `Z3.c` que vous n'avez pas à modifier, et où vous placerez vos fichiers `main.c` et `Solving.c`
- Un répertoire `examples` contenant deux codes de programmes vous montrant comment manipuler les graphes et comment interagir avec Z3.
- Un exécutable (compilé au CREMI, qu'il est donc sage de n'utiliser que là) `workingEqualPath` qui représente ce qu'on attend d'un projet avec toutes ses améliorations (à ceci près qu'il n'affichera évidemment pas la formule qu'il utilise). Il pourra vous servir à tester la correction des réponses de votre programme.
- Un répertoire `graphs` contenant des instances qui vous sont proposées pour tester votre programme (et celui qui vous est fourni). Il contient un dossier `assignment-instances` contenant les 3 graphes du sujet précédent, un dossier `instances-prime-length` contenant trois graphes dont les chemins acceptant sont tous de longueur multiples de respectivement 3, 5 et 7 (testez ces graphes deux à deux), et un dossier `generic-instances`, contenant de nombreuses instances classées par satisifiabilité, puis par taille puis par numéro (chacun des ces sous-dossiers numérotés contient une instance complète, appelez vos programme sur le dossier en entier). Il n'est bien évidemment pas nécessaire de tout tester. Il est déconseillé de tester la version «formule globale» sur des instances dont les graphes ont plus de 30 nœud, car la formule commence à être très grosse. La formule par taille reste à peu près raisonnable.

Vous pourrez générer une documentation doxygen grâce à la commande `make doc`.

2 Travail à réaliser

Votre travail consiste à implémenter la réduction que vous aurez déterminé dans la première partie du sujet pour résoudre le problème Distance commune. À ce titre, vous devrez créer les fichiers main.c et Solving.c. Dans ce dernier, vous implémenterez les fonctions décrites dans Solving.h. Vous pourrez bien évidemment y définir autant de fonctions locales que vous jugerez nécessaires (simplement, elles ne pourront pas être appelées dans le programme principal).

Votre code devra être commenté dans le style des fichiers .h fournis, notamment pour les fonctions auxiliaires que vous implémenterez. La qualité des commentaires, du code, et en particulier la non-duplication de code seront considérés à l'évaluation. Tout projet ne compilant pas ne sera pas corrigé.

Étapes attendues :

— Minimum syndical : implémenter les fonctions getNodeVariable, getPathFormula et getFullFormula ainsi qu'un programme principal les utilisant pour résoudre le problème en explorant chaque longueur indépendamment. Il pourra afficher la formule générée, ainsi que les graphes parsés via un système d'option.

— Pour avoir la moyenne : En plus, récupérer le chemin calculé et l'afficher sur le terminal, en implémentant la fonction printPathFromModels.

— Amélioration 1 : Implémenter getFullFormula, et modifier le main de manière à obtenir uniquement une réponse au problème (oui/non).

— Amélioration 2 : Modifier getFullFormula de manière à ce que si la formule admet un modèle, il contienne uniquement les chemins témoins de la solution. Vous aurez également besoin d'implémenter getSolutionLengthFromModel.

— Amélioration 3 : Implémenter createDotFromModel pour afficher les chemins solution au format .dot (comme dans l'exécutable fourni). Il devra notamment contenir les informations de sommets sources et destination. Vous pouvez utiliser les couleurs de votre choix, mais nous serons content si vous respecter le même code couleur que nous.

— Amélioration 4 : Produire un exécutable avec un comportement similaire à celui fourni, c'est à dire capable de proposer les différents algorithmes et affichages via un système d'option.

L'amélioration 3 étant indépendante des 2 premières, elle peut évidemment être réalisée avant celle-ci, et un programme n'implémentant que celle-ci sera tout de même apprécié positivement. De même, si votre programme n'implémente pas toutes les améliorations 1 à 3, vous pouvez implémenter la quatrième en n'incluant que les options que vous avez implémentées.

3 Format des graphes

Pour vous épargner le travail consistant à écrire un parseur de graphe et à créer une structure de donnée graphe, vous permettant ainsi de vous focaliser sur l'efficacité de votre réduction, nous vous fournissons un parseur de graphes fournis au format .dot vers une structure de donnée `graph`. Après parsage d'un graphe, vous obtiendrez un objet de type `graph` le représentant, que vous pourrez manipuler à l'aide des fonctions détaillées dans le fichier `Graph.h`. L'implémentation exacte du graphe en mémoire n'est pas très importante de votre point de vue, vous les manipulerez *uniquement* via ces fonctions.

Le parseur prendra en entrée un fichier de graphe au format .dot. Le parseur qui vous est fourni supporte actuellement uniquement des graphes orientés, qui sont ceux que le projet utilisera. Plusieurs ensembles de graphes sur lesquels exécuter votre programme vous est fourni dans le dossier `graphs`. Vous pouvez évidemment créer vos propres exemples, et l'une des amélioration attendue du programme à réaliser consistera à créer des fichiers .dot affichant la solution. Pour ce faire, vous pouvez vous inspirer de la syntaxe des exemples fournis. La syntaxe complète du langage dot est décrite sur cette page. Vous pouvez afficher les graphes au format .dot grâce à `xdot` ou générer un fichier image le contenant grâce à `dot`.

4 Solveur SAT

Le solveur que vous allez utiliser est `Z3`, un solveur SMT développé par Microsoft Research. La page principale du projet est ici, et vous pouvez, si vous le souhaitez, vous familiariser avec cet outil là. Cependant, il n'est absolument pas nécessaire que vous suiviez ce tutorial, d'autant qu'il va bien plus loin que ce que nous utiliserons. Z3 est en effet un SMT solveur, c'est-à-dire qu'il est capable de manipuler des formules de la logiques du premier ordre. Nous n'utiliserons que la partie SAT-solving de Z3. Z3 dispose également d'une API C ce qui vous permettra de construire et de déterminer la satisfiabilité d'une formule de la logique propositionnelle directement depuis votre programme.

Pour ce qui vous concerne, un main exemple manipule les fonctions dont vous aurez besoin, et le fichier `Z3tools.h` contient une explication des fonctions que vous aurez à utiliser dans le cadre de ce projet (et qui sont des encapsulation de l'interface C de Z3).