Rapport du projet IA-IHM

PAPASIDERO Florian
L3Info

Paramètres de la fonction recuitSimule(...) : l'espace d'exploration S

Description générale

J'ai fait le choix d'une classe `CircuitGraphe` prenant en paramètre de constructeur le graphe dans lequel va se trouver le circuit élementaire que l'on veut traiter. De la manière dont est implémentée cette classe, on peut y créer des circuits hamiltoniens ou non, du moment qu'ils sont élémentaores.

Le circuit en lui-même est représenté par une liste (`PElement`) de `Sommet` implémentant l'attribut `v` via le type `InfoSommetCarte` déjà utilisé en TP. En effet, la classe `CircuitGraphe` est la seule classe que j'ai eu à créer puisque le problème du voyageur de commerce était déjà préparé via les TP précédents, faisant qu'il ne me restait plus qu'à utiliser toutes ces classes.

Du fait de l'utilisation de données dynamiques. Le constructeur par copie, l'opérateur = et le destructeur ont été implémentés explicitement. Les 2 premiers créent une copie dans le tas de la liste de sommets et copie le pointeur de graphe. Le destructeur lui, utilise la méthode statique de `PElement`, `efface1` pour supprimer la liste sans supprimer les sommets qui appartiennent au graphe. L'opérateur string et l'opérateur << ont également été mis en place.

J'ai d'abord pensé à représenter la liste des arêtes que l'on parcourra (pour calculer le coût) plutôt que la liste des sommets mais je me suis rendu compte que l'on faisait beaucoup de changements dans cette liste pour que cela soit plus intéressant qu'une liste de sommets.

La liste de sommets

Les sommets de la liste eux-mêmes ne sont pas créés dans cette classe. En effet, chaque élément de la liste (qui lui, est créé dans le tas par la classe `CircuitGraphe`) ne fait que pointer sur un sommet déjà existant dans le graphe (que la classe a reçu en paramètre).

L'utilisateur de la classe `CircuitGraphe` n'a pas à manipuler directement la liste, la classe facilite la gestion de la mémoire puisque tout est compris dans la classe.
Pour ajouter un sommet (en queue forcément) à cette liste, il faut donc passer un sommet qui est présent dans le graphe à la fonction `add(Sommet<InfoSommetCarte> *)`, qui fera également la vérification que cet ajout est possible (sommet non `NULL` présent dans le graphe, et absent de la liste) sinon une exception est levée.
Pour supprimer un sommet, on utilise la méthode `remove(Sommet<InfoSommetCarte> *)` qui lèvera une exception s'il n'y a rien à supprimer ou si le sommet n'est pas présent dans la liste.

Utiliser des exceptions "à tout va" dans `add` et `remove` est une manière de faire que je trouve un peu lourde mais elle a l'avantage d'être expressive sur l'erreur particulière commise, et pour le coup, je n'avais pas de meilleure idée.

Pour une bonne utilisation de cette liste dans la méthode `changementAleatoire`, il faut y mettre plus de 3 sommets (induit par le sujet). On doit également faire attention à ajouter les sommets (via `add`) dans l'ordre dans lequel on souhaite faire le parcours car c'est cet ordre qui sera utilisé par `coutParcours` pour récupérer les arêtes des sommets.

Paramètres de la fonction recuitSimule(...) : la solution initiale

Puisque le graphe utilisé dans le main est complet, je prend simplement la liste de tous les sommets du graphe dans l'ordre pour ma variable `CircuitGraphe` et j'y applique la fonction `changementAleatoire` pour obtenir un parcours quelconque. Et voilà ma solution initiale qui ne sera jamais la même à chaque exécution.

Paramètres de la fonction recuitSimule(...) : fonction coût

La fonction coût de la classe `CircuitGraphe` est implémentée sous le nom `coutParcours`. Elle récupère l'arête de chaque paire de sommets consécutifs dans la liste de sommets via la méthode `Arete<S,T> * getAreteParSommets(const Sommet<T> * s1, const Sommet<T> * s2) const` de la classe `Graphe` pour en extraire le coût. Si l'arête n'existe pas (graphe non complet), comme conseillé par le sujet, on estime qu'il y a une arête de longueur infinie, ce qui écartera la solution immédiatement dans le recuit simulé. La librairie standard C++ fournit `<limits>` pour créer un nombre infini.

Paramètres de la fonction recuitSimule(...) : fonction changementAléatoire

L'algorithme proposé par le sujet est utilisé :

1. On choisit deux sommets non consécutifs **A** et **B** au hasard dans ce cycle.

Pour cela, on a une méthode privée `Sommet<InfoSommetCarte> * sommetAleatoire() const` dans `CircuitGraphe` qui utilise un générateur de nombres (réellement) aléatoires fournit par la librairie standard C++ via `<random>` pour aller chercher un sommet de la liste.
Ensuite, dès que les conditions pour ces 2 sommets (différents et non consécutifs) sont remplies, puisque l'on doit changer le sens d'un certain nombre d'arêtes, et donc l'ordre de certains sommets, on crée un tableau dans le tas à partir de la liste pour faciliter cette manipulation.

2. On remplace l'arête **A->C** par l'arête **A->B**.
3. On change le sens du chemin **C->B**.
4. On remplace l'arête **B->D** par l'arête **C->D**.

Ceci est fait dans la méthode statique privée
```
static void nouvListeCircuitGraphe(Sommet<InfoSommetCarte> * s_preums, Sommet<InfoSommetCarte> *
s_deuz, CircuitGraphe & cg, Sommet<InfoSommetCarte> * tab_sommets[], const int size);
```
qui va remplir la liste de sommets du `CircuitGraphe` que l'on va renvoyer.

nouvListeCircuitGraphe

2. On remplace l'arête **A->C** par l'arête **A->B**.

On a juste à avancer dans le tableau en ajoutant chaque sommet jusqu'au premier de nos 2 sommets aléatoires inclus (**A**), on retient la position du suivant de ce sommet (**C**).

3. On change le sens du chemin **C->B**.

On avance jusqu'au deuxième sommet (**B**), on retient la position du suivant de ce sommet (**D**) et on repart dans le sens inverse du tableau en ajoutant chaque sommet jusqu'à **C** inclus.

4. On remplace l'arête **B->D** par l'arête **C->D**.

Pour finir, on se positionne à **D** et on ajoute tous les sommets jusqu'à la fin du tableau.

fonction main(...)

Pour tester la solution au TSP à l'aide de la fonction recuitSimule(), la carte utilisée est celle du TP3 pour aller plus vite, c'est donc un graphe complet. Je n'ai pas eu le temps de créer d'interface graphique donc je crée simplement le graphe et l'affiche dans la console, il est également enregistré dans un fichier. Enfin, concernant la solution du TSP, elle est affichée en console à la fin du recuit simulé.

Compilation

Testé sous une machine virtuelle Debian Jessie, il suffit d'un simple `make` pour créer les différents exécutables du projet (main + fichiers de tests). `make clean` pour supprimer les .o et `make cleanall` pour supprimer les .o et .out