UNIVERSITE DE CAEN BASSE-NORMANDIE

Rapport BVNS

M2 DECIM : Métaheuristique

Guillaume Dequesnes et Florian Benavent

10/02/2015

Rapport du devoir de Métaheuristique de M2 DECIM pour l'année 2014-2015.

Table des matières

Choix des modèles

Dans le cadre du projet, nous avons implémenté l'algorithme BVNS, ainsi que 4 mouvements.

L'algorithme en lui-même est simple et est basé sur le cours, dont sa définition était la suivante :

Algorithme 6: Steps of the Basic VNS

```
Function RVNS (s, k_max, t_max);
begin
    repeat
        k ← 1;
        repeat
            s' ← Shake (s, N_k) /* Shaking */ ;
            s'' ← FirstImprovement (s') /* Local search */ ;
            NeighbourhoodChange (s, s'', k) /*Change neighbourhood*/ ;
        until k = k_max;
        t ← CpuTime ()
    until t > t_max;
end
```

Pour les mouvements nous en avons choisi 4, dans l'ordre suivant : 5-Flip, Swap, 2-exchange et KempeChain.

5-Flip

Nous avons choisi de sélectionner jusqu'à 5 variables en conflits, ou le nombre de conflits si celui est inférieur à 5).

Pour chacune des variables ainsi tirées, on change leur valeur aléatoirement.

Swap

Pour le swap, nous sélectionnons au hasard une variable en conflit, et nous échangeons sa valeur avec la variable suivante, ou précédente s'il n'existe pas de variable.

Cette seconde variable n'a pas besoin d'être en conflit.

2-Exchange

Dans le 2-exchange, nous sélectionnons 2 variables, différentes l'une de l'autre.

La première est tirée aléatoirement dans l'ensemble des variables en conflits.

La seconde est tirée dans l'ensemble total des variables et peut donc, ou non, être en conflit.

Les valeurs de ces 2 variables sont ensuite échangées.

Kempe Chain

Pour le Kempe Chain, nous sélectionnons un nœud de départ aléatoire, et sélectionnons, à partir de lui, un composant connexe, ayant soit la couleur de la variable initiale, soit la première autre couleur rencontrée, tel que présenté dans le cours.

Pour chaque variable ainsi sélectionnée, nous inversons leurs couleurs.

Implémentation dans incop

Prise en main d'incop

La compréhension de la librairie au niveau général a été assez simple grâce à l'utilisation de la documentation doxygen associée.

Cependant sa compréhension au niveau des sources a été un peu plus compliquée du au découpage peu clair des fichiers, principalement pour les fichiers incop.h, incopalgo et incoputil qui aurait mérité d'être redécoupé, ce qui aurait facilité la compréhension du code de certaine classe, telle Configuration.

Le second problème, largement évoqué en TP, correspond aux problèmes de compilation. Ces problèmes sont de différentes natures, on y retrouve des includes manquantes (tel iostream, cstream, vector…), des using manquantes (principalement std, pour l'utilisation de std ::cout et std ::endl), et enfin des includes du C rejetés par les compilateurs c++ (tel <fstream.h>) .

Ces erreurs ont été corrigées en détail pendant les TP, pour les fichiers utilisés, et d'une manière plus large pour l'ensemble de la librairie, via CodeBlocks, pour permettre une compilation complète de la librairie.

Développement au sein d'incop

Dans un premier temps, nous avons choisi d'intégrer notre code directement dans les fichiers de la librairie, incopalgo.cc et incopalgo.h, de cette manière ils nous étaient de mettre en relation les différents objets dont on s'est inspiré dans la librairie.

A la fin du projet, après avoir eu un code fonctionnel et suffisamment testé, nous avons choisi d'extraire notre code et de l'inclure dans 2 fichiers, intitulés basicvns.cc et basicvns.h, avant de modifier le makefile, pour y introduire une nouvelle règle, basicvns.o, qui est appelé pour la génération de l'exécutable colorcsp.

Notre travail se divise finalement en 3 parties, les modifications appliquées à incop, l'algorithme BVNS en lui-même et les mouvements implémentés.

Modification d'incop

À part les modifications destinées à la compilation et la correction de problème au sein d'incop, la principale modification réalisée à la librairie est tout simplement l'inclusion de la génération de notre BVNS dans le fichier incoputil.cc. Pour ce faire on a simplement rajouté une fonction de parsing pour les paramètres, et une fonction d'initialisation, cette partie ayant été discutée pendant les TP, nous ne rentrerons pas dans les détails.

Les autres modifications concernent surtout l'ajout de commentaire destiné à l'éclaircissement de certaines parties d'incop.

Algorithme BVNS

Notre class BVNS implémente la classe IncompleteAlgorithm, pour respecter et être intégrer à la librairie.

Cette classe dispose de 6 attributs, son constructeur et la fonction centrale.

Les 2 premiers sont en charge de l'utilisation des mouvements.

```
// Maximum number of movements to use (we take the kmax first elements of the movement list)
int kmax;
// List Of Movement
std::vector<AbstractNeighborStructure*> movements;
```

Le premier attribut, kmax, est utilisé pour déterminer combien de mouvements parmi la liste movements seront utilisés.

La liste movements contient l'ensemble des mouvements que l'on autorise pour le BVNS. Cette liste est initialisée au constructeur et contient, dans l'ordre, 5-Flip, Swap, 2-exchange et KempeChain.

Les 2 suivants servent tout simplement pour le critère d'arrêt temporel de l'algorithme

```
// Max time required. Algorithm will stop when currentTime - startTime will be superior or equal to this value.
int maxTime;
// Start time of the BVNS Algorithm
time_t startTime;
```

Enfin, les attributs restants servent durant l'exécution de l'algorithme :

```
// Previous configuration of your problem;
Configuration* previous;
// Random walk algorithm
LSAlgorithm* walkalgo;
```

Le constructeur est uniquement en charge de l'instanciation des 2 variables configurables (kmax, et maxTime), ainsi qu'à l'initialisation de la liste des mouvements.

La dernière fonction, au cœur de notre class, correspond tout simplement à l'implémentation de l'algorithme. Reprenant le fonctionnement du Basic VNS tel qu'il a été étudié en cours.

La définition de la classe BVNS est présente dans le fichier basicvns.h.

Le code de l'algorithme est présent dans le fichier basicvns.cc

Mouvements.

Chacun des 4 mouvements choisis, présentés dans la première section de ce rapport, possède sa propre classe.

Ses classes disposent toutes d'un constructeur, vide, excepté pour le PFlip, qui récupère et la valeur p requise, pour l'associer à l'attribut correspondant.

Tous les mouvements implémentés doivent hériter de la classe AbstractNeighborStructure, définissant la fonction shake, pour pouvoir être utilisés par l'algorithme.

Le code en lui-même est évidemment assez similaire entre chaque mouvement, se passe en 3 temps.

Tout d'abord on sélectionne les variables sur lesquels on va appliquer le mouvement, avec les tests associés (éviter la sélection d'une même variable pour le 2-exchange par exemple).

Ensuite nous créons un nouveau mouvement via notre problème, que nous remplissons avec les variables présélectionnées.

Enfin nous appliquons ce mouvement sur notre problème.

La définition de chaque classe est présente dans le fichier basicvns.h.

Le code des 4 mouvements implémentés est présent dans le fichier basicvns.cc

Script

Dans le but d'offrir une utilisation facile du programme, et pour effectuer des séries de test efficaces, nous avons choisi d'ajouter un ensemble de scripts.

Le premier, build.sh, est en charge, comme son nom l'indique, de la compilation des sources. Il s'agit au final d'un simple appel au makefile modifié d'incop, mais nous avons préféré ainsi regrouper les tâches de compilation/exécution au niveau supérieur des sources.

Le second script, run.py, est un script réalisé en Python, inspiré de celui fourni pendant les séances de TP, qui est en charge du lancement d'une exécution du programme.

Ce script est donc en charge de l'ensemble de la configuration du programme, mis à part le nom du fichier d'entrée et le nombre de couleurs autorisé.

Le nom de sortie est basé sur le nom du fichier d'entrée, mais au lieu d'être écrit dans le dossier examples, il est écrit dans le dossier out.

Son lancement se fait donc de la manière suivante :

```
?> python run.py « nom du fichier d'entrée » « nombre de couleurs autorisé »
```

Les 4 scripts restants sont en charge du lancement automatique d'une série de tests. Au début de chacun, le script build.sh est appelé, pour assurer l'utilisation de la dernière version.

Nous retrouvons donc les scripts :

- runDSJC.sh, chargé du lancement du programme pour les 12 fichiers d'exemples de la catégorie DSJC.
- runflat.sh, chargé du lancement du programme pour les 6 fichiers d'exemples de la catégorie flat.
- runle.sh, chargé du lancement du programme pour les 11 fichiers d'exemples de la catégorie le.

Le dernier script, runall.sh, s'occupe simplement du lancement dès 3 précédents, dans le même ordre que celui de leur présentation.

Résultats

Nous avons testé notre programme sur les jeux de tests fournis lors de la 1er séance de TD.

Nous dénombrons donc 12 jeux de tests de la série DSJC, 6 flat et 11 de la série le.

Nous avons effectué les tests pour l'ensemble de ces fichiers avec une même configuration, celle fournie par défaut par le script run.py, et 4 valeurs de couleurs pour chaque fichier.

L'ensemble de nos tests a donc été effectué avec une limite de 30 secondes. Notre algorithme cherche donc soit à optimiser jusqu'à la fin, soit, si une solution ayant 0 conflit est trouvée avant, s'arrête.

Nous avons choisi cette limite, relativement courte, pour favoriser le nombre de tests, en étudiant la vitesse de diminution des conflits comme facteur d'évaluation de notre algorithme, plutôt que de réaliser des tests très longs et peu nombreux, pouvant nous induire en erreur à cause de la marche aléatoire.

Dans les tableaux suivants sont indiqués, pour un problème et un nombre de couleurs donné, le nombre d'arcs en conflit restant au bout de 30 secondes.

Si ce nombre est à 0, il est indiqué également au bout de combien de temps la solution a été trouvée.

Série DSJC

Nombre de couleurs	5	6	7	9
DSJC125.5	300	234	188	129
DSJC250.5	1 344	1 073	887	630
DSJC500.5	5 716	4 640	3 884	2 894
DSJC1000.5	23 571	19 309	16 313	12 344

Nombre de couleurs	9	10	11	13
DSJC125.9	316	276	243	199
DSJC250.9	1 379	1 226	1 099	904
DSJC500.9	5 845	5 210	4 698	3 929
DSJC1000.9	23 966	21 462	19 428	16 273

Série Flat

Nombre de couleurs	20	21	22	24
flat300_20_0	299	275	263	219

Nombre de couleurs	26	27	28	30
flat300_26_0	188	164	455	120

Nombre de couleurs	28	29	30	32
flat300_28_0	155	135	98	117

Nombre de couleurs	50	51	52	54
Flat1000_50_0	1 458	1 400	1 390	1 328

Nombre de couleurs	60	61	62	64
Flat1000_60_0	1 171	1 139	1 089	1 087

Nombre de couleurs	76	77	78	80
Flat1000_76_0	819	782	799	745

Série le

Nombre de couleurs	5	6	7	9
le450_5a	346	209	127	11
le450_5b	356	224	124	3
le450_5c	611	367	150	0 -> 8.54 secondes
Le452_5d	613	308	130	0 -> 17.4 secondes

Nombre de couleurs	15	16	17	19
le450_15b	45	25	9	0 -> 2.64 secondes
le450_15c	277	229	205	148
le450_15d	275	232	206	148

Nombre de couleurs	25	26	27	29
le450_25a	0 -> 9.42 secondes	0 -> 0.99 secondes	0 -> 0.27 secondes	0 -> 0,12 secondes
le450_25b	0 -> 3.75 secondes	0 -> 0.43 secondes	0 -> 0.13 secondes	0 -> 0.08 secondes
le450_25c	84	68	57	31
le450_25d	89	75	62	34