TP2 Outils d'aide à la décision

Heuristiques pour le Job Shop

ESCOURBIAC Maxime, SEPTIER Jean-Christophe
ISIMA • 10 novembre 2010

Plan du compte-rendu:

I Modélisation du graphe

```
// Structure représentant un arc
struct arc{
  sommet * orig;
  sommet * dest;
  int duree;
};

// Structure représentant un sommet
struct sommet
{
  int id_job;
  int id_machine;
  list<arc*> entree;
  list<arc*> sortie;
  float distance;
  int sommet_restant;
  sommet * prec;
};

// Structure représentant t un graphe
struct graphe
{
  vector<sommet *> V;
  vector<arc *> E;
};
```

Notes:

On a choisi cette solution pour sa modularité et sa simplicité à l'application des algorithmes de ce tp.

On a rajouté des attributs dans la structure sommet afin d'accélérer l'algorithme de Bellman.

II-1 Génération du graphe

Les instances sont de la forme: (2 = nombre_job, 3 = nombre_machine)

2 3

0 34 2 45 1 34

2 87 1 23 0 56

On les lis, dans un fichier selon l'algorithme suivant:

ouverture du fichier;
lecture nombre_job,nombre_machine
allocation graphe;
création sommet initial & sommet puit;
pour i de 0 à nb_job faire
| prec = sommet initial;
| pour j de 0 à nb_machine faire
| | création nouveau_sommet;
| | lecture numero_machine, duree_opération;
| | création arc entre le nouveau_sommet et le sommet prec;
| | prec = nouveau_sommet;
| fait;
| creation arc entre le sommet puit et le sommet prec;
fait;
fermeture fichier;

Notes:

- Les créations prennent en compte allocations + adjonctions dans la structure de donnée graphe et sommet
- Les sommets sont rangés dans le vecteur V situé dans graphe, en sorte qu'on puisse retrouver facilement n'importe quel sommet via une fonction.
- V[0] est le sommet initial et V[1] est le sommet puit

II-2 Algorithme de Bellman

Cet algorithme permet de calculer le chemin le plus long, donc d'évaluer le makespan d'une instance. il permet aussi d'obtenir le chemin critique qui sera nécessaire pour l'algorithme de recherche locale

initialisation des sommets;
Pour tous les sommets faire
| recherche d'un sommet avec tous ses précédents marqués
| marquer le sommet;(on décrémente la variable sommet_restant pour chaque successeur)
| Pour tous les successeurs du sommet faire
| | mettre à jour la distance si elle supérieure;
| fait;
| //on recherche désormais le précédent de ce sommet pour avoir le chemin critique
| tant que j<nombre_prédécesseur && sommet->distance - prédécesseurs == 0 faire
| | passer a un autre prédécesseur;
| | ++j;
| fait
| mettre a jour le prec du sommet;
fait

Notes:
-Pour des raisons d'économie on n'effectue pas le chemin critique dans cette algorithme car on peut utiliser Bellman, juste pour obtenir le makespan d'une instance, mais on donne tous les outils pour le récupérer (parcours de liste chaînée)
-En performance pure cette implémentation de cet algorithme, pour une instance 10 machines 50 jobs et un vecteur de Bierwith aléatoire, on est en dessous du 10^-3s .
(PC personnel).

II-3 Recherche Locale

Le but de la recherche locale est d'améliorer le makespan grâce à la modification successive d'un vecteur de Bierwith passé en paramètre. cette procédure réalise une amélioration de type descente.

```
procédure recherche_locale(vecteur,itermax)
|  associer le graphe au vecteur;
|  évaluer le graphe; (Bellman)
|  génération du chemin critique;
|  prec = puit->prec; cour = prec->prec; //init des elts d'analyse du chemin critique
|  tant que fin == false && iter < itermax faire
|  |  tant qu'on se situe sur le même block machine faire
|  |  |  parcourir le chemin critique
|  |  fait;
|  |  Si on est sur le sommet initial alors
|  |  |  fin = true; //plus d'amélioration possible;
|  |  Sinon
|  |  |  re-formation du vecteur de Bierwith en inversant prec et cour;
|  |  |  associer ce nouveau vecteur au graphe initial
| |  |  evaluer le graphe
|  |  |  Si le makespan est meilleur alors
|  |  |  |  on genere le chemin critique
|  |  |  |  on re-init les elts d'analyse du chemin critique
|  |  |  Sinon
|  |  |  |  on annule la modification du vecteur;
|  |  |  |  on incrémente les éléments d'analyse du chemin critique ;
|  |  |  fsi;
|  |  fsi;
|  |  iter++;
|  fait;
fait:
```

Notes:

L'association d'un vecteur de Bierwith et du graphe se fait à l'aide de la méthode ajoutArcVecteur(int *), on peut aussi dissocier le graphe du vecteur avec supArc()
Ces méthodes sont valables pour les autres algorithmes

II-4 Algorithme mémétique

```
//initialisation de la population
tant que population.taille() < TAILLE_POP faire
|  générer vecteur aléatoire;
|  faire une recherche local avec le vecteur aléatoire et le graphe
|  Si vecteur est diffèrent de chaque vecteur de la population alors
|  |  insérer vecteur;
|  fsi;
fait;
trier la population selon le makespan;

//phase d'amélioration itérative
tant que iter < itermax faire
|  choisir 2 chromosomes parmi la population (choix élitiste && les chromosomes doivent
                                                être différents)
|  faire un croisement avec ces 2 chromosomes pour obtenir 2 chromosomes fils;
|  faire recherche_locale pour le fils 1;
|  effectuer une opération de mutation pour le fils 1;
|  Si le makespan du fils1 muté > makespan du fils 1 après recherche_locale  alors
|  |  chromosome_to_insert = fils1_muté;
|  Sinon
|  |  chromosome_to_insert = fils1;
|  fsi;
|  Si le makespan du chromosome_to_insert < plus mauvais makespan de la population alors
|  |  supprimer plus mauvais chromosome
|  |  insérer dans la population chromosome_to_insert;
|  fsi;
|  trier la population selon le makespan;
|  faire recherche_locale pour le fils 2;
|  effectuer une opération de mutation pour le fils 2;
|  Si le makespan du fils2 muté > makespan du fils 2 après recherche_locale  alors
|  |  chromosome_to_insert = fils2_muté;
|  Sinon
|  |  chromosome_to_insert = fils2;
|  fsi;
|  Si le makespan du chromosome_to_insert < plus mauvais makespan de la population alors
|  |  supprimer plus mauvais chromosome
|  |  inserer dans la population chromosome_to_insert;
|  fsi;
|  trier la population selon le makespan;
```

Note:

On se sert de la méthode unique des list pour éliminer les chromosomes identiques dans la population.

II-5 Algorithme génétique

```
//initialisation de la population
tant que population.taille() < TAILLE_POP faire
|   générer vecteur aléatoire;
|   associer le vecteur et le graphe;
|   évaluer graphe;
|   dissocier le vecteur;
|   Si vecteur est diffèrent de chaque vecteur de la population alors
|   |   insérer vecteur;
|   fsi;
fait;
trier la population selon le makespan;

//phase d'amélioration itérative
tant que iter < itermax faire
|   tant que population Fils.taille() < TAILLE_POP faire
|   |   choisir 2 chromosomes parmi la population (choix élitiste && les chromosomes doivent
|   |                                                    etre différents)
|   |   faire un croisement avec ces 2 chromosomes pour obtenir 2 chromosomes fils;
|   |   effectuer une opération de mutation pour le fils 1;
|   |   Si le makespan du fils1 muté > makespan du fils 1 après recherche_locale  alors
|   |   |   chromosome_to_insert = fils1_muté;
|   |   Sinon
|   |   |   chromosome_to_insert = fils1;
|   |   fsi;
|   |    inserer dans la populationFils chromosome_to_insert;
|   |    effectuer une opération de mutation pour le fils 2;
|   |   Si le makespan du fils2 muté > makespan du fils 2 après recherche_locale  alors
|   |   |   chromosome_to_insert = fils2_muté;
|   |   Sinon
|   |   |   chromosome_to_insert = fils2;
|   |   fsi;
|   |   inserer dans la population chromosome_to_insert;
|   fait;
|    trier la population Fils selon le makespan;
|    concaténer la population avec la populationFils
|    Supprimer les doublons de la population
|    Supprimer les plus mauvais chromosomes pour réatteindre la TAILLE_POP
fait;
```

Note:

Les populations sont stockées dans des listes pour pouvoir la trier plus rapidement grâce à la méthode sort() et supprimer les doublons grâce à la méthode unique().

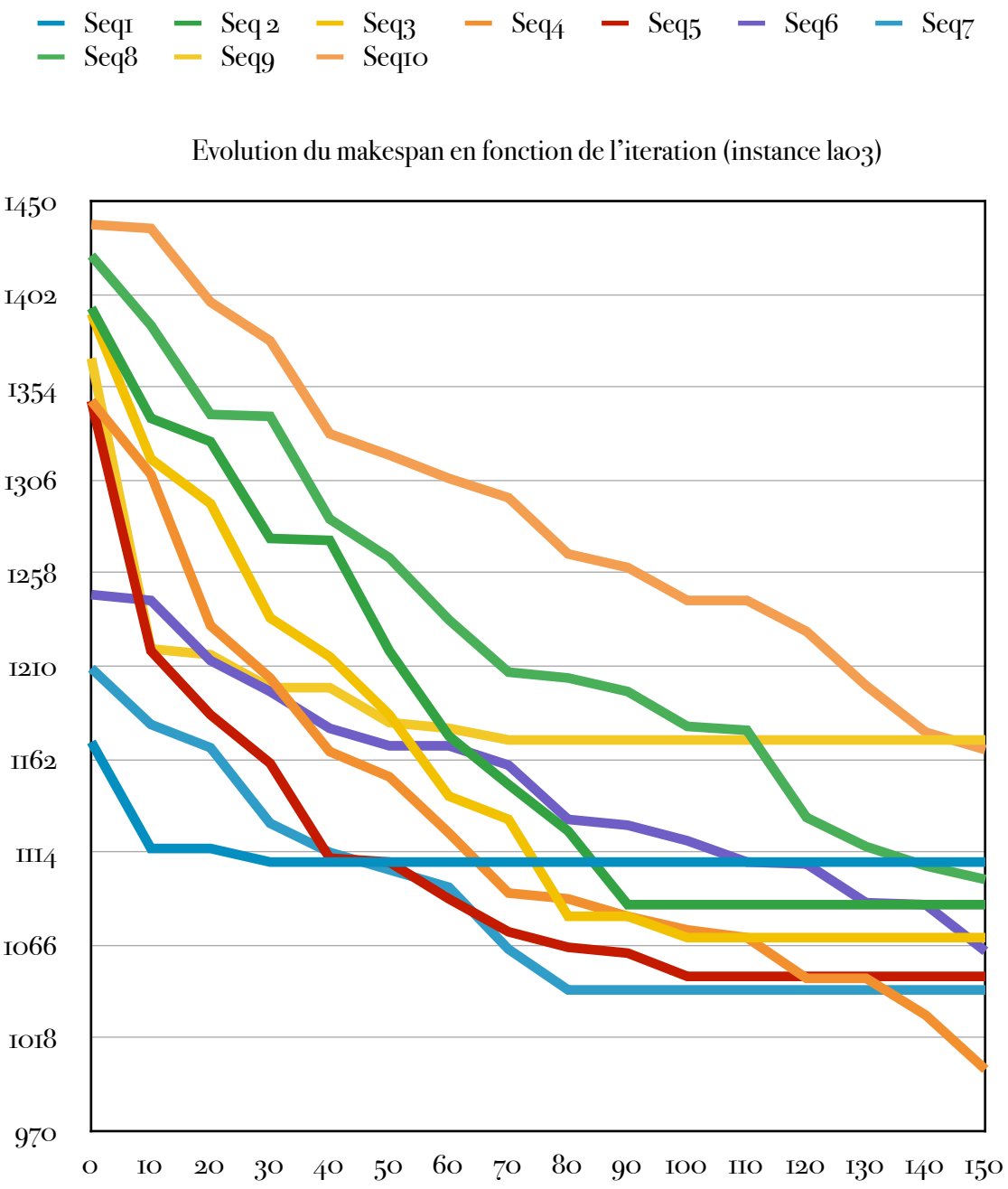
III-1 La recherche locale

Les essais sur la convergence de la recherche locale se fera avec l'instance la03 et avec des vecteurs de Bierwith qui seront généré aléatoirement.

On a exécuté 10 recherches locales.

Sur ce graphique, on a gardé que la condition d'arrêt sur le nombre d'itération

C'est à dire, dans le cas réel la recherche locale aurait pu s'arrêter avant le 150 itérations



III-2 l'algorithme mémétique

Les tests sont effectués de la sorte:

Pour une instance donnée on effectue 30 fois l'algorithme mémétique, on récupère ainsi le meilleur makespan parmi les 30.(optimum : valeur minimale théorique réalisable)

Pour toute les instances, on a configuré les variables suivantes:

Taille_population : 40 Itermax algo mémétique:500 Itermax recherche locale:50

Instance	NbJob	NbMach	Makespan	Optimum	Time/run
la01	10	5	666	666	0.8s
la02	10	5	665	655	0.9s
la03	10	5	617	597	0.8s
la04	10	5	593	590	0.9s
la05	10	5	593	593	1.0s
la06	15	5	926	926	1.4s
la07	15	5	890	890	1.5s
la08	15	5	863	863	1.4s
la09	15	5	951	951	1.3s
la10	15	5	958	958	1.2s
la11	20	5	1222	1222	1.8s
la12	20	5	1039	1039	1.7s
la13	20	5	1150	1150	1.8s
la14	20	5	1292	1292	1.9s
la15	20	5	1230	1207	2.0s
la16	10	10	978	945	1.6s
la17	10	10	973	784	1.7s
la18	10	10	859	848	1.5s
la19	10	10	873	842	1.6s
la20	10	10	907	902	1.8s

III-3 l'algorithme génétique

Les tests sont effectués de la sorte:

Pour une instance donnée on effectue 30 fois l'algorithme génétique, on récupère ainsi le meilleur makespan parmi les 30. (optimum : valeur minimale théorique réalisable)

Pour toute les instances, on a configuré les variables suivantes:

Taille_population : 40 Itermax algo génétique:500

Instance	NbJob	NbMach	Makespan	Optimum	Time/run
la01	10	5	666	666	1.0s
la02	10	5	675	655	1.1s
la03	10	5	619	597	1.1s
la04	10	5	606	590	1.0s
la05	10	5	593	593	1.0s
la06	15	5	926	926	1.8s
la07	15	5	890	890	1.7s
la08	15	5	863	863	1.7s
la09	15	5	951	951	1.6s
la10	15	5	958	958	1.7s
la11	20	5	1222	1222	2.2s
la12	20	5	1039	1039	2.0s
la13	20	5	1150	1150	2.0s
la14	20	5	1292	1292	2.1s
la15	20	5	1256	1207	2.3s
la16	10	10	972	945	1.8s
la17	10	10	983	784	1.7s
la18	10	10	861	848	1.9s
la19	10	10	866	842	1.8s
la20	10	10	912	902	1.8s

IV Conclusion

Analyses des résultats:

côté performance: l'algorithme génétique est plus performant que l'algorithme mémétique Lors des tests effectués, le temps d'exécution varie ,selon la taille de l'instance, et les paramètres dans les #define, c'est compris entre 0.8s et 2.0s pour le génétique et entre 1s et 2.3s pour le mémétique.

Par contre, côté résultat, l'algo mémétique grâce à la convergence de la recherche locale permet d'obtenir une solutions plus proche de la solution optimale quand elle ne le sont pas.

Bilan:

L'approche de type d'algorithme est intéressant, et le fait d'imiter un phénomène naturel pour réaliser des calculs de bonnes performances, est quelque chose pour nous d'innovant. Cependant, on peut approfondir ce tp en modifiant le type de descente,plus fort gradient etc.. Pour observer si on améliore la convergence vers la solution optimale.

Résultats:

Tous les résultats, et les traces sont disponibles dans l'archive envoyé par mail.