INF8215

TP2 - Optimisation du placement de micro-puces

Nom d'équipe: IA

Auteurs:

Lamia Salhi 2164386

Theo Delbecq 2161923

Professeur:

Quentin Cappart

Gaël Reynal

2020-2021

Contents

1 Introduction

Nous souhaitons optimiser le positionnement des composants sur une carte-mère. Le problème est modélisé de la manière suivante, on a : un ensemble de composants $C = \{1, ..., n\}$ et d'emplacements $L = \{1, ..., n\}$, une matrice de flux $flows_{ij}, \forall (i,j) \in C^2$ et une matrice de distance symétrique $dists_{kl}, \forall (k,l) \in L^2$. Une solution est un agencement attribuant à chaque machine un emplacement unique sans répétition. L'espace des solutions est l'ensemble des permutations $\phi \in S_n$ telles que $\forall i \in C, \varphi(i)$ est l'emplacement attribué au composant i. Par conséquent, la contrainte de différence est dure, il s'agit donc d'optimisation pure. La fonction objectif, qui est aussi notre fonction d'évaluation, est la suivante :

$$\min_{\varphi \in S_n} \sum_{(i,j) \in C^2} flows_{ij}\ dists_{\varphi(i)\varphi(j)}$$

Ce problème est de la famille du Quadratic Assignment Problem (QAP). C'est un problème NP-difficile nous avons donc essayé de le résoudre de manière approchée en implémentant plusieurs métaheuristiques.

2 Présentation des modèles et démarche

2.1 Initialisation et optimisation

Pour les différentes métaheuristiques que nous présentons, on utilise une initialisation glouton qui assigne à chaque composant un emplacement en fonction des coûts avec les composants déjà assignés. Cette initialisation s'est révélée un peu meilleure que l'initialisation purement aléatoire tout en laissant un degré de diversification. D'autres méthodes explorées sont présentées en annexe 1.
Plusieurs optimisations d'implémentation on été faites avec Numpy. Notamment la fonction d'évaluation ainsi que les matrices de distance et de flux ont été représenté par des tableaux Numpy afin d'accélérer les opérations matricielles, les masquages et les recherches d'extremums.

2.2 Tabu search

On commence par réaliser une recherche tabou car c'est une méthode simple qui donne a priori de bons résultats (mieux que le simulated annealing [3],[5],[6]).
Le voisinage utilisé est le 2-exchange : on intervertit l'emplacement associé à deux composants. Ce choix de voisinage permet de considérer l'ensemble des voisins comme valides car il reste peu coûteux. On exclut le 3-exchange (et les k-exchange supérieurs) qui n'apporte pas a priori de meilleur résultat pour le coût additionnel [6]. A chaque étape on choisi le meilleur voisin (pas nécessairement améliorant).
Pour la liste tabou, on utilise une implémentation en dictionnaire de temps plus optimisée car elle permet des vérifications en O(1). Les clés sont tous les couples composant/emplacement et les valeurs sont la dernière itération à laquelle ce couple a été assigné. Un voisin est tabou si les deux composants ont déjà été associés

à leurs nouveaux emplacements trop récemment. Nous avons fait ce choix car il permet d'être un peu moins restrictif que bannir un échange donc d'explorer plus de voisins sans être trop libre pour éviter les cycles et il est plus simple et rapide dans l'implémentation. De plus, ce choix a donné de bien meilleur résultat pour la méthode suivante.

Pour ne pas rater une bonne solution et permettre l'intensification quand elle est excellente nous avons mis un critère d'aspiration : si le voisin est meilleur que la meilleure solution jamais rencontrée il est sélectionné.

Le nombre d'itération durant lequel une assignation est tabou est un nombre aléatoire $t \in [0.9n, 1.1n]$ et il change toutes les $2.2n$ itérations. Ce choix permet d'être plus robuste sans avoir besoin de fixer ce paramètre ce qui soumettrait les résultats à plus de variations aléatoires [5].

2.3 Iterated tabu search (ITS)

Le choix de voisinage induit des améliorations lente car les déplacements sont petits dans l'espace de solution et la recherche tabu ne permet donc pas de s'éloigner beaucoup de la solution de départ. On constate une convergence vers l'optimal lente voire même non atteinte pour B et C alors que l'on peut réaliser plus de 300 000 itérations. Afin de limiter l'errance et la localisation autour de mauvaises solutions on ajoute un mécanisme d'Iterated Local Search qui permet de faire un plus grand saut lorsque l'on stagne sans repartir au hasard et perdre l'intensification faite jusqu'alors [2].

Lorsque les résultats de la recherche tabou ne s'améliorent pas pendant un certains nombres d'itérations on perturbe la solution et on relance une recherche. Deux perturbation on été testées : une aléatoire ou l'on échange 10% des affections aléatoirement et une gloutonne ou l'on replace les 10% retirés avec l'assignation qui minimise le coût. La perturbation gloutonne a donné de meilleurs résultats en moyenne. Afin de mitiger l'aspect d'intensification on repart de la solution en cours et pas de la meilleure solution à chaque restart.

Une perturbation a lieu si le résultat stagne durant $10n$ itérations (ce paramètre s'est montré plus performant que 7, 30, n, 2*n, 20*n, 100*n). On conserve la liste tabou.

Sur ITS le choix des éléments de la tabou list notamment affecte beaucoup les résultats :

Meilleurs résultats de ITS selon élément de la tabou list					
	requete A	requete B	requete C	requete D	requete E
Moves tabou*	58	3683	13399	171266	255270
Assignations comp/empl tabou **	58	3683	13178	170769	255183

*Moves tabou : on bloque les échange. Exemple : $\varphi(i) <-> \varphi(j)$ bloqué directement

**Assignations comp/empl tabou : on bloque les assignations. Un échange est interdit s'il mène à deux assignations interdites simultanément. Exemple : $\varphi(i) <-> \varphi(j)$ bloqué si $(i, \varphi(j))$ et $(j, \varphi(i))$ sont bloqués

Travailler en bloquant les assignations plutôt que les mouvements reduit la restriction et permet une plus grande liberté lors de la convergence.

2.4 GRASP

Un défaut de la recherche tabou est qu'elle reste localisée et ITS est une solution à ce problème. Les initialisations étant assez faible et potentiellement bloquantes nous avons implémenté GRASP afin d'avoir une façon différente et plus aléatoire d'explorer l'espace à partir de bonnes solutions.

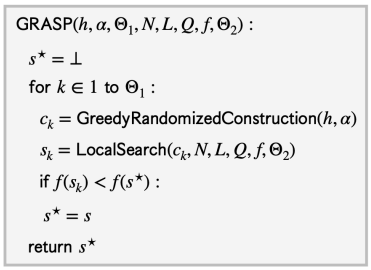

Figure 1: Pseudo code Grasp (pris du cours INF6102 de Quentin Cappart, Polytechnique Montréal)

La construction utilisé considère tous les couples composants/emplacements à chaque étape et sélectionne une assignation parmi celles qui minimisent le coût partiel par rapport aux assignations déjà faite. Le paramètre α détermine la part des meilleures assignations considérée à chaque étape relativement au meilleur coût possible. Avec h le coût partiel d'une assignation, on retient les assignations qui respectent :

$$h(assignation) \leq h_{min} + \alpha(h_{max} - h_{min})$$

La sélection se fait selon une roulette pour favoriser les meilleurs assignations car la force de cette méthode d'initialisation était déjà limitée.

La recherche utilisée ensuite est la tabu search avec une limite de 30 secondes. L'idée de cette limite et que l'on souhaite tout de même explorer plusieurs solutions notamment afin d'utilsier GRASP comme algorithme d'initialisation et d'avoir plusieurs solutions générées.

On implémente une version réactive de grasp pour augmenter la robustesse par rapport au paramètre alpha. Entre chaque itération un alpha est choisi par roulette parmi la liste arbitraire suivante : [0, 0.1, 0.2, 0.3, 0.4, 0.5]. On ne va pas jusqu'à 1 pour garder un choix favorables aux meilleurs éléments dans la construction. Les probabilités sont mise à jour selon les performances moyenne obtenues avec chaque alpha. Les variations étant faible par rapport à la moyenne les différents α on souvent une chance assez équivalente d'être choisi, cependant on ne change pas cela pour garder de la diversification car la roulette de l'algorithme de construction est déjà rééquilibrée par standardisation et normalisation ce qui favorise déjà l'intensification.

2.5 Algorithme génétique

Nous avons également réalisé un algorithme génétique. Celui se compose en plusieurs étapes : Tout d'abord, il y a l' **initialisation** de la population que nous faites avec des solutions générées aléatoirement sur une population correspondant à 20*(le nombre de composants), ce qui permet une grande diversification tout en

permettant la convergence dans la limite de temps de calcul fixée à 20 minutes pour chaque instance. Ensuite, l'algorithme génétique itère une série d'étapes sur plusieurs génération. Le **nombre de génération** est fixé à 1000 arbitrairement mais elle ne sera jamais atteinte dans nos tests dû à nos deux autres critères d'arrêt présentés plus tard.

Voici les étapes que nous itérons :

- la **phase de sélection** pour sélectionner les solutions élites de notre population sous le format d'un tournoi. C'est à dire que chaque individu de la population est le meilleur parmi 60% des individus de la population initiale choisi aléatoirement. En effet nous avons choisi cette taille de tournoi (0.6*taille de la solution) pour assurer une bonne intensification.

- la **phase d'hybridation**, ou crossing pour générer des solutions enfants à partir des parents. Pour cela, on part de 2 parents et on génère 2 enfants avec une probabilité défini par le "crossing rate" en utilisant le "uniform crossover". Ce dernier crée un masque binaire de même taille que les solutions, puis, le premier (second) enfant est défini en conservant les valeurs du parent 1(parent 2) là où le masque a des valeurs égale à 1(0). Le reste est défini dans le sens de lecture, en considérant les valeurs du parent 2(1) n'étant pas déjà dans l'enfant 1(2) (toujours dans le sens de lecture). Voici un exemple :

mask=[0,1,0,0,1,0]	
Parent1 = [1,3,4,2,5,6]	Enfant1 = [**6**,3,**2**,**1**,5,**4**]
Parent2 = [3,6,2,1,5,4]	Enfant2 = [3,**5**,2,1,**6**,4]

Nous avons choisi le UX car il permet une plus grande diversification pour deux parents différents toutefois il donne des parents peu varié lorsque les parents sont proches. Le cross_rate est défini de façon adaptatif, d'après le papier [1] pour de grande population on aurait une meilleur convergence pour un mutation rate croissant défini par $(i/NB\_GENERATIONS)$ avec i la ième génération. Ce paramètre est couplé au "mutation rate" défini dans la phase de mutation.

- la **phase de mutation**, modifie nos solution avec un certain taux(**"mutation rate"**). Une loi de poisson choisi aléatoirement le nombre **k** d'éléments à modifier dans une solution (nombre au minimum égale à 2 dans notre cas pour conserver l'unicité des valeurs dans les solutions). Une fois ce nombre k fixé, on choisi uniformément les éléments à modifier dans la solution enfant et on réalise une permutation entre eux. Le paramètre λ de la loi de poisson est fixé à 3 au début pour avoir une convergence contrôlé et au bout de 30 d'itérations sans améliorations du résultats on le défini aléatoirement à $int(N\_COMPONENTS * r.uniform(0.01, 0.2))$ puis, après 60 itérations à $int(N\_COMPONENTS * r.uniform(0.01, 0.3))$ sans améliorations. On espère pouvoir amener de la diversité en laissant une plus grande gamme de valeur pour le λ tout en laissant la possibilité à l'algorithme de converger. Enfin, le mutation rate est défini de façon adaptatif [1] avec des valeurs décroissantes égalent à $(1 - i/NB\_GENERATIONS)$ avec i la ième génération. Cela permet une bonne diversification durant les premières générations et un intensification croissante au cours des générations.

- la **phase de mise à jour du meilleur résultat**. On enregistre la meilleure solution de la population

actuelle si elle est meilleure que la meilleure solution déjà enregistré précédemment.

- la **phase de mise à jour de la population** où on garde les 10% meilleures solutions de la population parent et 90% de la population enfant pour intensifier la recherche. De plus à cette étape on supprime les solutions clones et on en génère de nouvelles aléatoirement pour conserver la taille de notre population. Supprimer les clones permet une meilleure diversité.

Le **critère d'arrêt** utilisé est le temps limité à 20 minutes et une stagnation du score sur 100 itérations d'affiler. Pour favoriser nos chances d'obtenir le minimum optimal nous avons également réalisé des **restarts**. Voici un tableau récapitulatif de tous les paramètres utilisés:

paramètres de l'algorithme génétique						
Taille population	nb génération	taille du tournoi	cross rate	mutation rate	λ	% de parent
20*nb_components	1000	0.60*Taille population	croissant	décroissant	3*	10%

*comme expliqué plutôt le lambda est adaptatif

La principal difficultés de l'algorithme génétique se trouve dans le choix des paramètres qui sont nombreux et qui dépendent les uns des autres et nous ne pensons pas avoir les valeurs optimales à ces paramètres.

2.6 algorithme génétique + GRASP

Enfin, nous avons tenté une méthode hybride où nous utilisons les résultats du GRASP pour initialiser notre algorithme génétique. En effet, en faisant plusieurs tests sur l'algorithme génétique on sentait que l'initialisation était un point clé et que les résultats obtenus en dépendent fortement. Par conséquent, on a tout d'abord testé plusieurs initialisations avec des solutions greedy et des solutions générés aléatoirement mais sans succès. Finalement, on a décidé d'initialiser une fine partie de la population en faisant en sorte que cette initialisation ne dure que 1 minutes dans notre recherche génétique quelque soit l'instance.

nombre d'individus initialisé par GRASP dans la population en 1 minute				
requete A	requete B	requete C	requete D	requete E
100/100	64/400	20 /600	6/1400	6/1600

3 Résultats et discussions

3.1 Résultats

Meilleurs résultats des métaheurisitiques par requête					
	requete A	requete B	requete C	requete D	requete E
naive search	120	3899	13816	173771	258574
Tabu Search	58	3722	13409	171292	255270
ITS	58	3683	13178	170769	254548
GRASP	58	3683	13178	170883	254782
Genetic Search	58	3683 (rarement)	13399	171086 (rarement)	255249
Genetic Search + GRASP	58	3683	13178 (rarement)	171125	254984
optimum	58	3683	13178	inconnu	inconnu

3.2 Discussion

Les résultats obtenus avec la recherche tabou sont déjà bien meilleurs que l'aléatoire mais l'optimal n'est pas trouvé dès l'instance B, alors que sa taille reste raisonnable et que l'on fait par conséquent plus de 300 000 mouvements en 20 minutes. De plus, on a pu constaté des variations sensibles entre plusieurs exécutions (dans le pire des cas 3789 pour la B). Cela montre que cette recherche est fortement soumise à son initialisation et à l'aléatoire, elle reste localement bloquée et parfois seule la chance la dirige correctement. Nécessairement, cet aspect se ressent sur les plus grandes instances ou l'on s'améliore moins par rapport à l'initialisation car on ne fait plus qu'une dizaine de millier de mouvements alors que l'espace est plus grand, on reste donc bloqué localement autour de l'initialisation.

ITS et GRASP ont de meilleurs résultats qu'une simple recherche tabou car ils permettent tous les deux de régler ces problèmes en offrant une option pour une meilleure exploration de l'espace, soit assez proche avec ITS, soit vers une nouvelle bonne solution avec GRASP. On obtient notamment l'optimum sur B et C. Les résultats de ces deux algorithmes sont assez équivalents et offrent la même amélioration par rapport à la recherche simple. ITS semble cependant parfois légèrement plus performant que GRASP. GRASP étant plus aléatoire avec une exploration basée sur l'heuristique d'initialisation retenue qui n'est pas très performante, on peut penser que l'intensification autour d'une solution est un meilleur choix qu'espérer partir d'une meilleure initialisation.

Cependant ces deux algorithmes font moins de différences avec une recherche tabou simple sur les grandes instances. On peut l'expliquer de deux manières : les opérations de voisinage étant bien plus coûteuses on fait beaucoup moins d'itération et de recherche tabou et l'espace étant plus grand on converge lentement avec le voisinage choisi. Les mécanismes de restart ne sont donc que rarement déclenché, quelques dizaines de fois, or, ceux-ci étant assez soumis à de l'aléatoire, on profite moins de leurs apports avec peu d'itérations. D'autre part, le temps utilisé pour ces restart est consommé par la recherche simple pour continuer son intensification et donc avec de la chance elle peut trouver une solution de qualité assez proche.

Pour l'algorithme génétique, on explore l'espace totalement différemment des méthodes précédentes. L'hybridation permet des déplacements plus importants à chaque étape tout en ramifiant une zone prometteuse de l'espace.

On trouve aussi l'optimum sur A et B. Cependant le minimum est très rarement trouvé pour l'instance B et varie entre ce minimum et 3760. Ce qui montre une grande variance et aussi une bonne intensification mais également que notre algorithme a du mal à s'échapper des minimums locaux et/ou prend trop de temps à s'en échapper. Par conséquent, notre algorithme est très dépendant de l'initialisation de notre population.
Les itérations de GRASP étant indépendantes, on décide de l'utiliser pour initialiser l'algorithme génétique et lui donner un meilleur départ. Dans ce cas, on a le minimum constamment pour la B contrairement ou Genetic Search seul et on arrive même parfois à obtenir le minimum pour la C. Globalement on a de meilleurs résultats avec l'initialisation GRASP qu'avec une initialisation aléatoire sauf dans de très rare cas comme pour l'instance D. L'algorithme génétique est plus lent à converger car pour trouver une bonne solution par hybridation, il a besoin de combiner des solutions déjà bonne, il ne peut pas s'y diriger directement. Par conséquent avec une initialisation aléatoire il lui faut le temps de se constituer une bonne base puis il commence à générer de bonnes solutions. L'initialisation GRASP permet de constituer plus rapidement cette base.
On note également que lors de l'entraînement le genetic search n'améliore que très peu les résultats donnés à l'initialisation par GRASP. Une explication possible est que pour une grande instance il est difficile d'atteindre l'optimum lorsqu'on s'en rapproche par hybridation car les modifications doivent être plus fines. Et, étant donné qu'un ensemble de solution aléatoire est aussi généré pour compléter la population générée par GRASP, l'algorithme génétique peut être amené à s'éloigner brusquement de l'optimum vers un autre minimum local ou composer des solutions moins bonnes. Ce qui fait que la population globale prend du temps à se diriger vers les solutions du GRASP.
Finalement, ITS propose les meilleurs résultats. Avec plus de temps, l'algorithme génétique pourrait l'égaler ou le dépasser. La lenteur de sa convergence à l'approche de l'optimum comme mentionné plus tôt, limite la possibilité d'avoir une meilleure solution en 20 minutes.

4 Conclusion

En conclusion ce TP nous aura permis de tester 3 algorithmes de recherche local qui sont la recherche Tabou, l'algorithme génétique et le GRASP. Si nous devions choisir une des méthodes on se tournerait vers l'ITS qui nous donne des meilleurs résultats en un temps raisonnable de 20 minutes pour toutes les instances. Cependant pour la génération de solutions multiples, GRASP a des résultats très proches et les mécanismes aléatoires permettent une meilleure robustesse notamment sur un temps plus court.

5 Références

[1] Hassanat, A.; Almohammadi, K.; Alkafaween, E.; Abunawas, E.; Hammouri, A.; Prasath, V.B.S. Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach. Information 2019, 10, 390. https://doi.org/10.3390/info10120390

[2] Misevicius, Alfonsas Lenkevicius, Antanas Rubliauskas, Dalius. (2006). Iterated tabu search: An improvement to standard tabu search. Information Technology And Control. 35. 10.5755/j01.itc.35.3.11770.
[3] Mickey R. Wilhelm Ph.D., P.E. Thomas L. Ward Ph.D., P.E. (1987) Solving Quadratic Assignment Problems by 'Simulated Annealing', IIE Transactions, 19:1, 107-119, DOI: 10.1080/07408178708975376
[4] Deshpande, Vivek Chopde, Dr. (2005). Facility Layout Design by CRAFT Technique.
[5] E. Taillard, Robust taboo search for the quadratic assignment problem, Parallel Computing, Volume 17, Issues 4–5, 1991, Pages 443-455, ISSN 0167-8191,
[6] Burkard, R. E., Dragoti-Cela, E., Pardalos, P. M., Pitsoulis, L. S. (1998). The quadratic assignment problem. In Handbook of Combinatorial Optimization (Vol. 2, pp. 241-337). Kluwer Academic Publishers.
[7] R. E. Burkard and U. Derigs, Assignment and Matching Problems: Solution Methods With FORTRAN-Programs, Berlin, Germany:Springer-Verlag, pp. 120-121, 1980

6 Annexe

6.1 Méthodes d'initialisation

Plusieurs méthodes d'initialisation par construction on été testées dans l'objectif de renforcer le départ des recherches :

- Aléatoire : On génère des permutations aléatoirement. Cela nous permet d'avoir une baseline mais aussi d'augmenter la diversification avec des restarts.
- Greedy 1 : On sélectionne à chaque étape le composant avec la plus grande somme de flux sortants et on l'associe à l'emplacement avec la plus petite somme de distance sortantes. En cas d'égalité, le choix est aléatoire. L'intuition est que l'on sélectionne en priorité les composants les plus importants et on les mets aux emplacements qui sont en moyenne les meilleurs.
- Greedy 2 : De la même manière qu'avec le critère précédent, on alloue cette fois à chaque étape directement un couple composant/emplacement en triant chaque couple selon le ratio $\frac{sommedesdistancessortantes}{sommedesfluxsortants}$ pour mitiger la priorité du critère de flux.
- Greedy 3 : La faiblesse des initialisations précédente est que l'on ne tient pas compte du coût réel généré par les associations emplacements/composants, on ne se base en quelque sorte que sur des moyennes. On essaye donc de se baser sur les coût partiels générés par chaque choix pour se rapprocher du coût réel. La procédure est la suivante :
  - On sélectionne le composant avec le plus de flux liés à des composants assignés.
  - En cas d'égalité on sélectionne le composant avec la somme de flux non assignés la plus faible (on limite l'impact des flux que l'on ne connaît pas). On sélectionne aléatoirement en cas d'égalité.

  - Pour le composant sélectionné on choisit l'emplacement qui génère le moins de coût par rapport aux composants déjà assignés.
  - En cas d'égalité, on sélectionne l'emplacement avec la somme des distances sortantes non assignées minimale.

- Greedy 4 : L'algorithme précédent est très restrictif et déterministe. Le risque en utilisant cette initialisation à répétition est de générer les mêmes solutions et d'être coincé dans un minimum local. On propose donc une version plus diversifié de la troisième initialisation greedy avec des random tie break plus adaptées à des initialisation de masse.

- Greedy statistique : Afin d'avoir une meilleure idée des coût réels généré par une affectation on tente une approche statistique. Pour chaque couple composant/emplacement, on génère des solutions aléatoire et on calcule le coût moyen. On prend l'assignation la moins coûteuse à chaque étape.

- Increasing degree of freedom : L'idée de cet algorithme est de se baser de la même manière sur les coût réels générés par les machine déjà attribuée cependant on autorise d'échanger une nouvelle assignation avec une déjà prise d'où l'idée de degré de liberté croissant [7].

L'algorithme CRAFT a aussi été envisagé mais s'est révélé trop coûteux [4].
Les diverses initialisations ne sont en pratique seulement un peu meilleure que sélectionner la meilleure solution d'un grand nombre d'initialisations aléatoire. Cela vient du fait que l'on ne peut connaître le coût de chaque élément d'une solution avant de l'avoir totalement généré et évaluer toutes les associations possibles pour chaque variables revient à résoudre le problème. Donc les initialisations par construction restent très approximatives. Pour profiter des aspects aléatoires de ces algorithmes, on génère 10000 solutions et on conserve la meilleure afin de partir d'une des meilleures solution atteignables. Générer plus de 10000 solutions ne donnait pas de meilleurs résultats et le temps d'initialisation reste assez cours pour les cinq premières méthodes.
La meilleure initialisation est la greedy 4. On peut expliquer le fait qu'elle dépasse les initialisations 1 et 2 car ses critère sont plus précis et proches de la réalité. Elle dépasse 3 car l'aspect trop déterministe ne permet pas de profiter des initialisations multiples. Greedy statistique est moins bon car il donne des moyenne de valeur pour chaque assignation mais cela ne renseigne pas vraiment sur le choix qui mène au minimum. Enfin Increasing degree of freedom demande un temps de calcul trop long sans apporter de gain.