Systèmes Multi-Agents

Octobre 2021

TP3 : Algorithmes d'appariement

Rapport

Crédit : StockSnap de Pixabay

Jean DESCAMPS
Vivien LECLERCQ

Première partie

Introduction

Dans ce rapport nous nous intéressons aux algorithmes d'appariement et de leurs implémentions. Nous présentons le travail que nous avons effectué lors des TD 3 et 4 de l'UV IA04 à l'Université de Technologie de Compiègne.

1. Problème d'appariement

Avant de commencer, il est impératif de comprendre l'objet du problème. Nous nous intéressons au problème d'appariement défini comme suit lors de la séance 3 du TD.

— Deux groupes de même taille sont composés d'agents.
— On cherche à apparier deux à deux ces agents de manière univoque.
— Chaque agent dispose de préférences quant aux agents de l'autre groupe.

Qu'est ce qu'une instance du problème ?
Une instance du problème d'appariement se caractérise par les entrées qui le composent. Ainsi lorsque la taille n des ensembles ou que les préférences des agents sont différents, il s'agit d'instances différentes de ce problème. Les algorithmes que nous avons vu ne traitent pas en particulier une instance du problème mais ils traitent l'ensemble des instances possibles. C'est-à-dire que les algorithmes s'adaptent aux données qu'il a en entrée pour proposer une sortie.

Quelle réponse est attendue ?
La réponse correspond à la sortie attendue du problème. Selon ce que l'on souhaite, il est possible d'ajouter certaines propriétés à la sortie du problème. En effet, dans certains cas il est nécessaire que l'appariement soit parfait ou alors que l'appariement soit stable. Ces conditions sur la sortie du problème vont être déterminantes pour définir les algorithmes qui vont répondre à ce problème. Nous avons vu dans le cours qu'il était préférable dans la réalité d'obtenir un appariement parfait et stable. Or, il serait possible dans ce problème d'obtenir des réponses seulement parfaites sans que cela ne pose de problème.

Structure de données pour l'appariement

Pour ce travail, nous avons fait le choix d'implémenter une structure de données "Mariages" correspondant au type suivant :

```
type Mariages map[AgentID]AgentID
```

La clé de ce map correspond à l'AgentID du disposant. La valeur correspond à l'AgentID du proposant. Cette représentation nous permet de représenter tous les appariements dans un ensemble non ordonné. L'avantage est aussi lié à la complexité du map comparé à des listes, la complexité est proche de $O(1)$ rechercher un élément dans un map alors qu'elle est de $O(n \times log(n))$ pour un tableau.

L'intérêt d'implémenter cette structure de données réside dans le fait que nous pouvons alors implémenter des méthodes nous permettant de vérifier la stabilité et le biais de nos appariements.

2. Génération d'ensembles de test

Dans ce projet, nous avons eu besoin de générer des instances aléatoire du problème d'appariement pour une taille n donnée.

Nous avons décidé de garder la structure Agent présente dans le code originel.

Nous avons généré des méthodes permettant de générer un objet Prenom qui correspond au résultat de l'extraction de toutes les données présentes dans le fichier Prenoms.csv fourni pour ce projet. Ces méthodes permettent de filtrer les noms extraits en fonction du genre ou bien de la langue. Il existe en particulier une fonction ShufflePrenoms qui permet de générer n noms aléatoires.

Ensuite, nous avons créé une fonction GenerateAgents dans le fichier agents.go qui permet de générer deux ensembles d'Agent de manière aléatoire. Cette fonction correspond au fait de générer une instance du problème.

3. Validation de la fonctionnalité de nos algorithmes

Dans une optique de maintenabilité du code et afin de vérifier que nos algorithmes sont fonctionnels, nous avons également implémenter des tests unitaires pour chacun de nos algorithmes afin de vérifier qu'ils terminent correctement, proposent un appariement adéquat et qu'ils soient stables (si l'algorithme de base doit produire un appariement stable comme pour Gale-Shapley)

Deuxième partie

Dynamique libre

1. Présentation de l'algorithme

L'algorithme de Dynamique Libre est une simulation du monde réel lorsqu'il n'y a aucune autorité responsable de l'appariement.

Il consiste à faire se rencontrer les couples et à résoudre itérativement les instabilités.

2. Implémentation de l'algorithme

L'algorithme Dynamique Libre se contente, d'initialiser des mariages aléatoire et de récupérer deux mariages de l'ensemble des mariages et de vérifier leur stabilité. La stabilité de deux mariages peut être rétablie à l'aide de l'algorithme Stabiliser Mariage.

3. Terminaison de l'algorithme

L'algorithme Dynamique Libre est un algorithme qui ne termine pas forcément. Nous avons eu beaucoup de difficulté à trouver une configuration d'agents particulière qui prouve la propriété précédente.

En effet, il suffit d'être capable de trouver une configuration pour laquelle l'algorithme ne termine pas pour prouver que l'algorithme ne termine pas dans tous les cas.

La recherche d'une telle configuration nous a pris beaucoup de temps. Il est clair intuitivement une telle solution existe : une solution où les préférences des agents sont incompatibles, où chaque mariage instable provoque une nouvelle instabilité. Nous avons d'abord essayé de trouver cette solution par la logique. Cependant, nous avons été incapable de la trouver.

Nous avons donc fixer la taille du problème à trois agents de chaque côté, ce qui fait un total de 6 agents. De plus, il existe 6 configurations d'ordre de préférences pour chaque agent étant donné que les préférences concernent les trois agents de l'ensemble opposé. Il existe donc $6^6 = 46656$ configurations possibles pour un tel problème. Nous avons donc implémenter un algorithme qui teste ces 46 656 possibilités, le but étant de trouvé une possibilité pour laquelle l'algorithme Dynamique Libre ne termine pas.

Pour ce faire, nous avons utilisé une fonction de génération de configuration aléatoire pour un taille de problème de données. Chaque configuration est stockée dans un tableau de configurations. Pour qu'une configuration soit tester, il ne faut pas qu'une configuration

identique soit déjà dans le tableau. Au final, les structures utilisées se trouvent dans le fichier *configuration.go*.

4. Stabilité de l'algorithme

L'algorithme de Dynamique Libre, lorsqu'il termine, donne toujours une solution stable.

La démonstration de cette propriété est intuitive car l'algorithme se termine justement lorsque la solution d'appariement est stable.

Il est possible de démontrer cela par l'absurde. Imaginons que l'algorithme ait produit une solution d'appariement instable. Cela équivaut à dire que l'algorithme s'est terminé en trouvant une solution instable. Cela est absurde car l'algorithme ne se termine que lorsque l'appariement est stable.

5. Biais de l'algorithme

L'algorithme de dynamique libre a pour biais l'égalité des agents. Chaque agent obtient la meilleure situation qu'il peut obtenir en maximisant le bien être global de tous les agents.

Contrairement aux autres algorithmes, il n'a y pas de différence de traitement entre les proposant et les disposant. La notion même de proposant est de disposant n'existe même pas car aucun agent ne fait de proposition ou ne les accepte. De ce fait, il existe une certaine égalité de traitement entre tous les agents. On pourrait considérer que le fait d'avoir une égalité de traitement équivaut au fait de ne pas avoir de biais particulier mais quelque part, d'un point de vue philosophique, le fait de ne pas avoir de biais est un bais en soi. Le biais est une condition nécessaire (qui ne peut pas ne pas être) pour les algorithmes.

Troisième partie

Algorithme d'Acceptation Immédiate - Boston

1. Présentation de l'algorithme

Un groupe joue le rôle des proposants, l'autre groupe joue le rôle des disposants. Les proposants envoient des propositions à leur disposant préféré. Ensuite, les disposants acceptent leur proposition préféré. L'algorithme s'arrête quand tous les proposants ont trouvé un disposant.

2. Implémentation de l'algorithme

Vous trouverez deux fonctions AcceptationImmediateAlgorithm et deleteAgents qui implémente cet algorithme de manière modulaire. Un type Proposition a également été introduit :

```
type Propositions map[AgentID][]AgentID
```

La clé est l'AgentID du disposant et la valeur un tableau d'AgentID qui correspondent aux AgentID des proposants qui ont fait des propositions.

3. Terminaison de l'algorithme

L'algorithme d'acceptation termine en $O(n^2)$ tours de boucle principale. En effet on considère au minimum un couple (proposant,disposant) par tour (cout $O(n) + O(n) = O(n)$), et le parcours total du tableau de proposant (pire des cas) coute egalement $O(n)$ ici, d'ou une complexite de $O(n^2)$. il produit également un appariement parfait (c'est la condition de sortie de boucle).

4. Stabilité de l'algorithme

L'algorithme de Boston ne produit pas toujours d'appariement stable. La fonction GenerateProblematicAgentsForAcceptationImmediateAlgorithm permet de générer une instance du problème pour laquelle l'algorithme donnera une solution instable.

5. Biais de l'algorithme

L'algorithme est très avantageux pour les proposants. En effet, les propositions des proposants ont plus d'importance. Si un disposant ne reçoit aucune proposition de son proposant préféré, alors il n'a aucune chance de pouvoir se marier avec lui.

Quatrième partie

Algorithme d'Acceptation Différée - Gale Shapley

1. Présentation de l'algorithme

L'algorithme d'acceptation différée est similaire à l'algorithme de Boston mais diffère dans l'acceptation par les disposants qui est maintenant temporaire : ils peuvent rejeter leur proposant actuel pour le nouveau. Il a été présenté pour la première fois en 1962 par David Gale et Lloyd Shapley. Cet algorithme termine ($O(n^2)$) et propose un appariement stable mais est fortement biaisé : les proposants partent du meilleur des mondes pour descendre graduellement et les disposants du pire vers le meilleur.

2. Implémentation de l'algorithme

La signature de la procédure implémentant cet algorithme est la suivante :

```
func AcceptationDifferee(agtA []*Agent, agtB []*Agent, a Mariages)
```

Le type agent a par ailleurs été modifié afin d'ajouter un attribut "itérateur" permettant d'avancer dans les préférences des proposants lorsque ces derniers sont refusés ou rejetés. Un map "engagés", local à la fonction, permet de vérifier lorsque l'on boucle sur les proposants si le proposant considéré est déjà apparié ou non. Dans le cas positif, on passe au proposant suivant. Sinon on regarde si le disposant n'est pas deja apparié, auquel on l'ajoute à l'appariement et on le valide dans le map local "engagés". Si le diposant est déjà apparié, on compare le nouveau proposant à l'ancien, dans le cas où le premier est préféré on supprime l'ancien et on ajoute le nouveau du map "engages" et on actualise l'appariement avec le nouveau. L'utilisation de cette méthode nous faisant passer par un map local n'est pas optimal. Elle nous oblige en effet à boucler sur l'ensemble des proposants à chaque tour de la boucle "tant que l'appariement n'est pas complet". Elle nous permet cependant de s'affranchir facilement de la gestion des proposants encore à considérer, sur ce point l'algorithme pourrait être amélioré.

3. Fonctionnement de l'algorithme, stabilité de l'algorithme

Dans le fichier de test main_test.go, nous implémentons un test unitaire qui nous permet d'apporter une première vérification du bon fonctionnement de notre implémentation pour

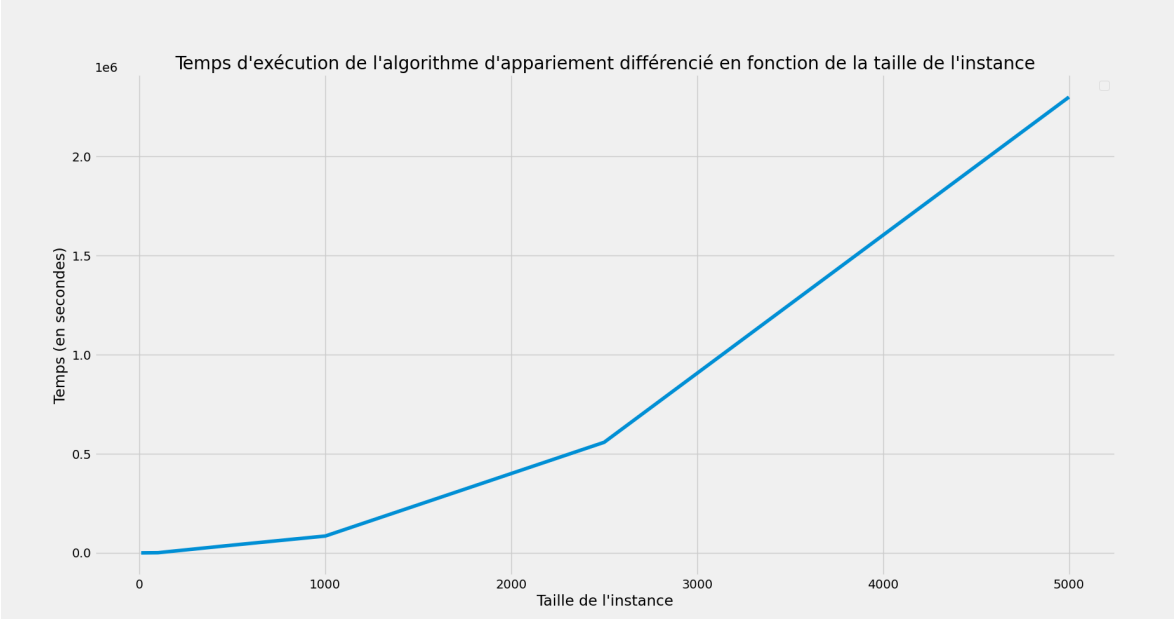
une petite instance du problème (n=4). Ce test unitaire nous permet également de vérifier la stabilité de l'appariement (ici implémenté pour n=50 mais modifiable)

4. Terminaison et correction de l'algorithme implémenté

Notre implémentation de l'algorithme est équivalente à celle proposée en langage naturelle :

- la boucle "tant qu'il reste un proposant" est équivalente à la boucle "tant que le nombre de couples l'appariement n'est pas égale au nombre de proposants initiaux"
- l'utilisation de l'itérateur nous assure que le proposant propose au disposant qu'il préfère parmi ceux à qui il n'a pas déjà fait de proposition.
- le bloc conditionnel de fin de fonction nous assure que le disposant abordé répond "non" si la proposition ne lui convient pas, ou bien "peut-être" si la proposition lui convient, en rejetant l'offre son partenaire actuel.

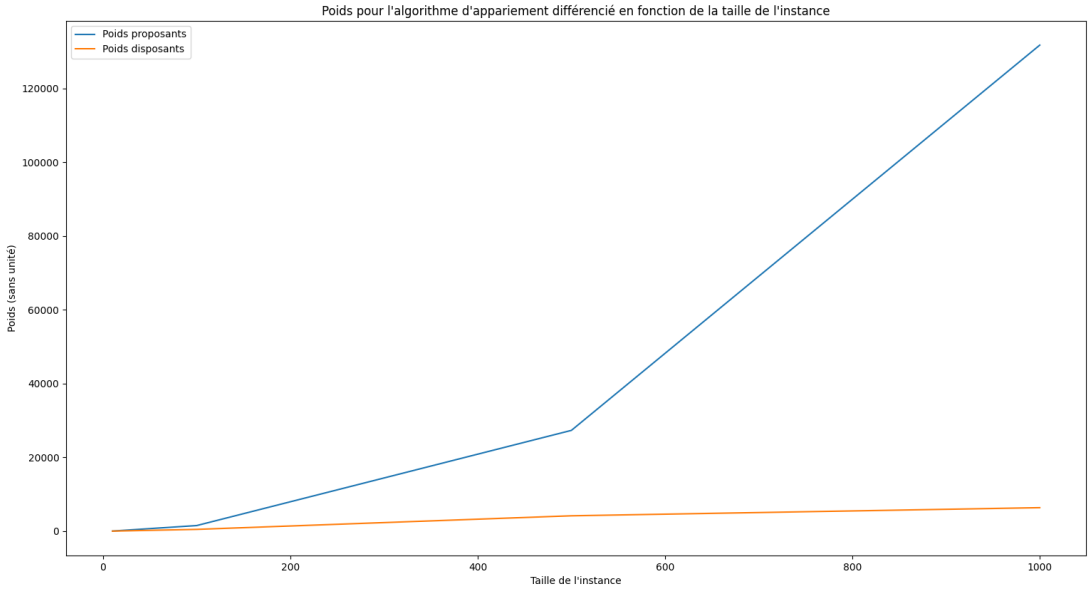
Les démonstrations effectuées en cours nous assurent alors que l'algorithme nous fournit un appariement parfait (c'est la condition de sortie de boucle) et stable (vérifié par le test unitaire). On a également vu que l'algorithme d'acceptation différée termine avec une complexite temporelle de $O(n^2)$: un même couple ne peut être considéré deux fois, et le cardinal du produit cartésien de l'ensemble des proposants par celui des proposants est de n^2 Empiriquement on retrouve bien un temps d'exécution quadratique pour l'algorithme que nous avons employé :



5. Biais de l'algorithme implémenté

Les démonstrations effectuées en cours nous assurent que l'algorithme d'acceptation différée favorise les proposants par rapport aux disposants : les proposants parcourent leur liste de préférence par ordre décroissant et par construction de la disposition, les disposants n'acceptent que des échanges qui leur sont favorables. La question du biais (groupe favorisé par rapport à l'autre) nous semblait plus difficilement abordable par l'intermédiaire d'un test unitaire, nous avons alors évalué celui-ci à l'aide d'une fonction comptabilisant le "poids". Nous entendons par "poids" la somme sur tous les proposants/disposants des indices dans les tableaux de preference du disposant/proposant considéré dans l'appariement. Ainsi plus la somme totale est élevée, plus il a fallu globalement descendre bas pour les proposants par exemple.

Une simulation pour différentes tailles d'instances donne le graphe suivant :



Il semblerait que l'evolution du poids par rapport à la taille du probleme soit quadratique pour les proposants et linéaire pour les disposants.

Cinquième partie

Algorithme de Cycles d'Echanges - Top Trading Cycles

1. Présentation de l'algorithme

L'algorithme : Top Trading Cycles (Gale - Shapley Scarf, 1974 - Abdulkadiroglu Sonmez, 2003) fonctionne de la manière suivante : Tant que tous les agents n'ont pas été appariés, les proposants pointent vers les disposants non-appariés qu'ils préfèrent. Les disposants pointés pointent alors vers leurs proposants préférés. On identifie ensuite un cycle d'échange et l'on apparie chaque agent du cycle avec l'agent qu'il pointe.

2. Implémentation de l'algorithme

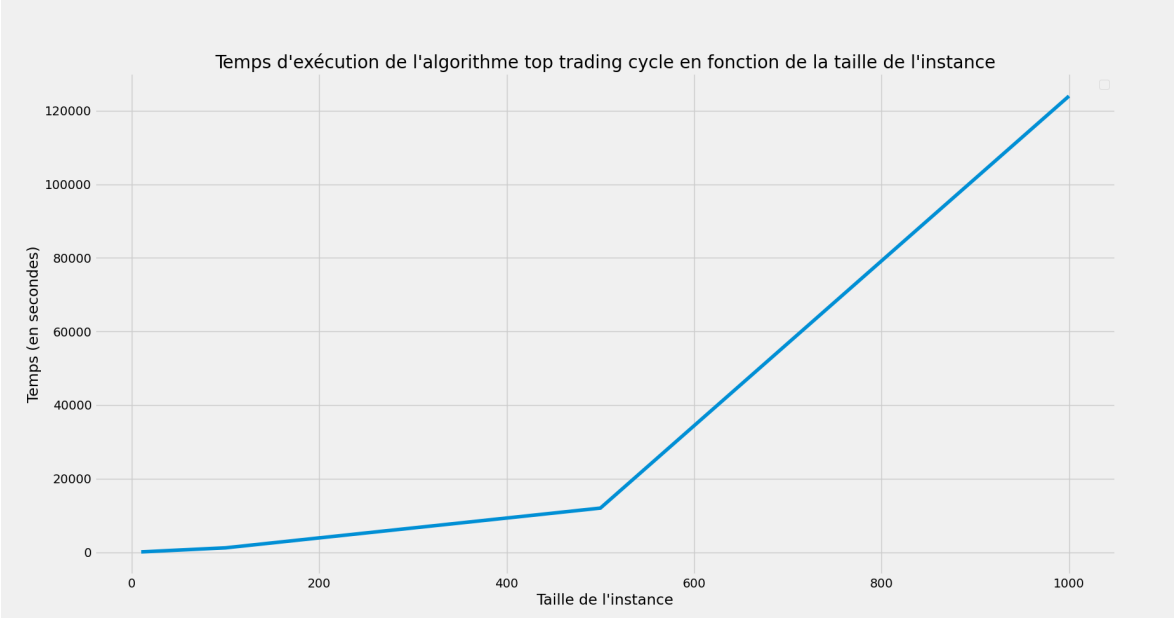
La signature de la procédure implémentant cet algorithme est la suivante :

```
func TopTradingCycles (agtA []*Agent, agtB []*Agent, a Mariages)
```

Comme pour le Gale Shapley, la condition de sortie de boucle est l'égalité entre la taille de l'appariement et le nombre de proposants (condition equivalente à : "tous les agents ne sont pas appariés.") Dans un premier temps on identifie les disposants ainsi que les proposants qui sont pointés à l'aide de deux map[AgentID]AgentID (pointeurs_disp et pointeurs_prop respectivement). On construit ensuite le cycle à l'aide de ces deux maps et on en extrait les couples qui doivent être appariés à l'aide d'une autre fonction.

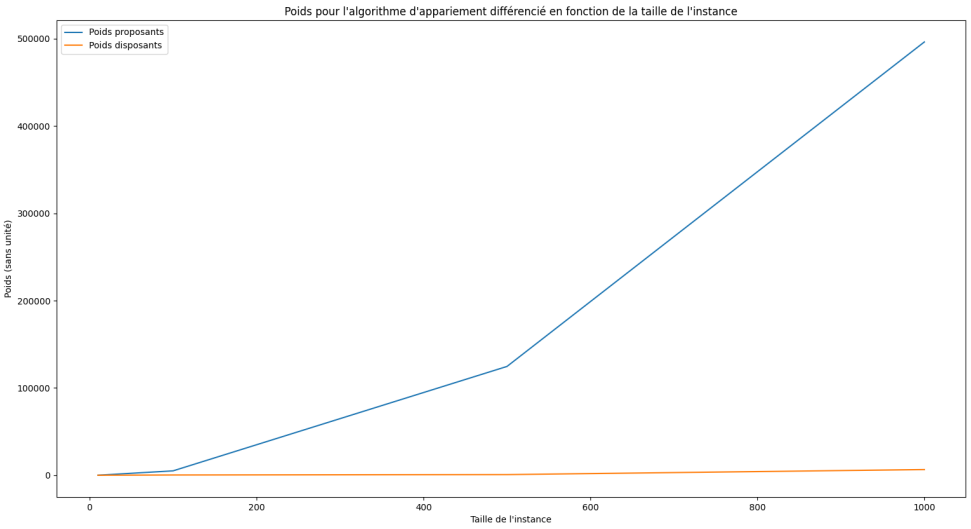
3. Terminaison et fonctionnement de l'algorithme

Cet algorithme termine également en $O(n^2)$. En effet à chaque iteration, il existe au moins un cycle de taille 1 minimale, on ajoute donc au mininum un couple dans l'appariement, la taille de l'appariement est donc une suite strictement croissante, on a donc l'assurance que la condition de sortie de boucle soit atteinte. Au pire des cas, L'algorithme fournit bien un appariement (a chaque iteration, on retire au minimum un couple disp/prop) parfait (c'est la condition de sortie de notre algorithme). Cet appariement est stable.



4. Biais de l'algorithme

De la même manière que pour le Gale Shapley, on regarde le poids pour l'ensemble des disposants et des proposants :



A nouveau, il semblerait que celui-ci evolue de maniere polynomiale pour les proposants et linéairement pour les disposants.