Rapport de projet OCaml/IA

Résolution de conflits aériens par Branch & Bound

COSTE Dorian

LI Zhen

POUGET Lilian

TOBELEM Sam

Encadrant : Richard Alligier

Table des matières

[Introduction 3](#_Toc534700866)

[Présentation du problème 3](#_Toc534700867)

[Notre organisation 4](#_Toc534700868)

[Première version : séparation primale 5](#_Toc534700869)

[Présentation de OCaml 5](#_Toc534700870)

[Présentation de l’algorithme *Branch and Bound* 5](#_Toc534700871)

[Benchmarking 6](#_Toc534700872)

[Deuxième version : changement de l’heuristique et de la fonction filtre 8](#_Toc534700873)

[Amélioration de l’heuristique : choisir les manœuvres par 2 8](#_Toc534700874)

[Amélioration de la fonction filtre 8](#_Toc534700875)

[Implémentation d’un filtre initial 9](#_Toc534700876)

[Benchmarking 9](#_Toc534700877)

[Analyse des résultats 9](#_Toc534700878)

[Conclusions 10](#_Toc534700879)

[Annexe : données benchmarking 11](#_Toc534700880)

[Bibliographie 13](#_Toc534700881)

# Introduction

## Présentation du problème

Nous choisissons le projet « Résolution de conflits aériens par Branch & Bound ». Le projet est mis en place pour nous permettre de confirmer notre maîtrise du langage de programmation fonctionnelle OCaml, et pour complémenter le cours sur les algorithmes d’Intelligence Artificielle *(IA)*.

Le problème que nous devons résoudre est, en soi, simple.

* Nous avons plusieurs avions, indiqués par la lettre i. Nos avions passent dans une zone de contrôle, et doivent tous s’éviter entre eux.
* Nous savons déjà quelles manoeuvres chaque avion peut effectuer (il y a 160 possibilités). Un modèle géométrique a permis de déterminer, pour chaque paire d’avions (i, j), quelles manoeuvres compatibles peuvent effectuer ces avions afin de conserver une séparation suffisante.

Par exemple, on sait que si l’avion 1 effectue la manoeuvre 160 (continuer tout droit), alors l’avion 2 peut effectuer uniquement les manoeuvres 1 à 20 (pas la manoeuvre 160 : il ne peut pas continuer tout droit).

* Nous connaissons également le coût des manoeuvres. La manoeuvre 160 a un coût nul.
* Nous voulons trouver la solution, qui est la meilleure combinaison des manoeuvres de chaque avion. Les manoeuvres choisies pour chaque paire d’avions doivent être compatibles, et la somme des coûts des manoeuvres doit être minimale.

Le problème se formume ainsi comme un problème d’optimisation combinatoire entière (Programmation Linéaire en Nombres Entiers, ou PLNE) :

Ici :

* est notre variable de décision, la combinaison des manoeuvres : , avec le nombre d’avions et le nombre de manoeuvres possibles.
* est la fonction objectif, la somme des coûts des manoeuvres choisies.
* est l’ensemble des contraintes : il décrit quels sont valides au regard de la compatibilité des manoeuvres de chaque paire d’avion.

Lorsqu’il y a peu d’avions et peu de manoeuvres possibles, on résout en testant chaque possibilité.

Prenons un exemple avec 3 avions et 3 manoeuvres possibles :

|  |  |  |  |
| --- | --- | --- | --- |
| **Avion 1 : Manoeuvre** | **Avion 2 : Manoeuvre** | **Avion 3 : Manoeuvre** | **Résultat** |
| 1 | 1 | 1 | Compatible : coût total 30 |
| 1 | 1 | 2 | Incompatible |
| 1 | 1 | 3 | Incompatible |
| 1 | 2 | 1 | Incompatible |
| 1 | 2 | 2 | Compatible : coût total 60 |
| 1 | 2 | 3 | Compatible : coût total 20 |
| 1 | 3 | 1 | Incompatible |
| ... ... ... ... | | | |
| 3 | 3 | 3 | Incompatible |

On trouve alors la solution, qui est le résultat *Compatible* de coût total minimal.

Le problème, c’est que nous voulons traiter de gros problèmes, avec 160 manoeuvres possibles et des dizaines d’avions. Pour 30 avions, notre tableau aurait ... lignes !

En effet, avec cette méthode de résolution, la complexité est exponentielle, en . Nous cherchons donc à optimiser la résolution du problème, pour qu’un ordinateur puisse l’effectuer en un temps raisonnable pour 30 ou 40 avions.

L’algorithme de *Branch and Bound* va nous permettre de supprimer des lignes à notre tableau, d’évacuer des possibilités qu’on sait ne pas être solution : nous n’aurons alors plus à considérer toutes les possibilités.

Au cours du projet, nous évaluons des solutions algorithmiques différentes, dans le but de pouvoir résoudre le problème en un minimum d’opérations possibles. Nous mettons donc en place ces solutions, et les testons pour voir lesquelles sont plus rapides selon les cas.

## Notre organisation

Dorian a tout de suite pris le projet en main. Lui et Lilian ont rédigé les grandes lignes de l’algorithme. Zhen a écrit des fonctions annexes (comme des fonctions de filtrage de solutions compatibles). Sam a aidé Zhen, et s’est concentré sur la rédaction du présent rapport.

# Première version : séparation primale

La première version de notre algorithme est notre *Mimimum Viable Product* : il doit … fonctionner.

Nous nous concentrons sur la mise en place de l’algorithme tout simple, car nous voulons rapidement avoir un algorithme qui fonctionne et qu’on pourra améliorer dans un deuxième temps.

## Présentation de OCaml

OCamll est un langage de programmation de niveau industriel supportant les styles fonctionnel, impératif et orienté-objet (ocaml.org)

Nous profitons surtout des fonctionnalités qu’offre le langage OCaml pour coder en programmation fonctionnelle (i.e. sans effet de bord).

Ce langage est adapté pour coder des algorithmes d’optimisation. L’avantage est que le code est facile à débugger car, à la compilation, toutes les erreurs de liaison entre les fonctions sont relevées : on peut donc manipuler plus facilement des types complexes (comme des *int list array*).

## Présentation de l’algorithme *Branch and Bound*

Un algorithme par séparation et évaluation, ou Branch and Bound en anglais, est une méthode générique de résolution de problèmes d'optimisation combinatoire. (Wikipédia)

L’algorithme de Branch and Bound ne se contente pas d’énumérer des solutions (*Branch*, ou séparation récursive du problème en plus petits sous-problèmes) : il borne (*Bound*) la solution.

On classe les manœuvres à choisir selon un heuristique : le coût minimal potentiel total de la solution, dans le cas du choix de la manœuvre en question. Il nous permet d’éviter de traiter un grand nombre de solutions.

Exemple : supposons que la manœuvre 2 est sélectionnée pour le premier avion, et que pour le deuxième avion, on sache :

* Si on choisit la manœuvre 1, le coût total sera au minimum 10
* Si on choisit la manœuvre 2, le coût total sera au minimum 20

*Avion 4*

*Avion 3*

*Avion 2*

*Avion 1*

Dans ce cas, nous commençons par séparer et considérer la manœuvre 1. Si, après traitement, on trouve une solution qui coûte 15, on n’aura même pas à considérer la manœuvre 2, car on sait qu’un tel choix nous donnera de toute manière une solution moins bonne que celle trouvée.

Le calcul de l’heuristique nécessite peu d’opérations : on somme les coûts des manœuvres compatibles les moins chères pour les avions restants (ici : les avions 3 et 4). Trouver une heuristique demande donc un temps constant (au maximum opérations si les manoeuvres sont déjà triées).

Notons que, dans le calcul de l’heuristique, la compatibilité entre les manœuvres à choisir sur les avions restants (3 et 4) n’est pas prise en compte : on prend en compte uniquement la compatibilité avec la manœuvre de l’avion 2. D’où le fait que l’heuristique n’est qu’une minoration : la solution « prendre toutes les manœuvres les moins chères sur les avions restants » n’est sûrement pas valide !

Pour rendre tous ces calculs plus rapides, nous avons fait en sorte que les manœuvres soient triées en coût croissant. C’est logique : nous voulons considérer en premier les manœuvres les moins coûteuses.

## Benchmarking

Le contenu fourni par Richard Alligier comporte plusieurs fichiers de benchmarking. Chaque fichier représente une situation, un problème, et contient des informations sur les trajectoires des avions : entre autres, la matrice de compatibilité des manœuvres entre chaque avion.

On teste donc notre algorithme sur différents fichiers, pour tester le temps qu’il met à trouver la solution. La valeur qui nous intéresse est la taille maximale d’un problème solvable en un temps raisonnable (i.e. quelques minutes).

Pour cette première version, on arrive à traiter en une seconde tous les fichiers décrivant des situations à 20 avions … Mis à part deux fichiers qui posent problème.

Notre programme nous donne également accès au nombre de nœuds qu’explore l’algorithme. Un grand nombre de nœuds explorés va de pair avec une inefficacité de l’heuristique : cela signifie que, plusieurs fois, l’heuristique a guidé l’algorithme vers l’exploration d’une branche de l’arbre qui s’est révélée trop coûteuse. L’écart entre l’heuristique et la réalité trompe l’algorithme, qui fait alors plus de recherches inutiles.

Nous cherchons à booster ces performances avec les améliorations présentées dans la partie suivante.

# Deuxième version : changement de l’heuristique et de la fonction filtre

## Amélioration de l’heuristique : choisir les manœuvres par 2

Dans l’exemple précédent, à chaque itération, nous choisissons pour l’avion suivant la manœuvre la moins chère. Cependant, nous pourrions éviter ce choix dans le cas où il est incompatible avec les autres manœuvres peu chères des autres avions.

En effet, si le choix la manœuvre la moins chère implique forcément le choix d’une manœuvre chère pour tous les autres avions, alors c’est un mauvais choix.

Pour tenter d’améliorer les performances, nous implémentons un nouvel heuristique qui, au lieu de sommer les coûts des manœuvres les moins chères pour chaque avion, somme les coûts des couples de manœuvres les moins chers pour chaque paire d’avions.

Ce changement implique plus de calculs pour trouver l’heuristique à chaque itération. En effet : plaçons-nous dans le cas où il reste 14 avions à instancier. L’heuristique naïf additionne les coûts des manœuvres les moins chères : il nécessite 14 recherches dans les tableaux des manœuvres compatibles restantes à chaque avion.

Le nouvel heuristique va dans un premier temps calculer, pour chaque couple d’avions, le couple de manœuvres le moins cher parmi ceux compatibles avec les choix précédents. En supposant qu’il reste 50 manœuvres compatibles à chaque avion sur 160, cela fait 50² = 2500 coûts de couples de manœuvres à calculer. Il y a , soit 91 paires d’avions : on arrive déjà à 227 000 opérations.

Mais ce n’est pas terminé : l’heuristique va maintenant comparer les coûts minimaux des 91 paires d’avions afin de sélectionner la paire d’avions la moins chère. On réitère ensuite, sans la paire d’avions trouvée, avec les 12 avions restants, afin de trouver la paire suivante la moins chère … jusqu’à avoir traité tous les avions.

Toutes ces opérations sont le prix à payer pour un heuristique plus rapide. Et les résultats paient.

## Amélioration de la fonction filtre

Notre tuteur, Richard Alligier, nous propose d’implémenter un nouveau filtre utilisant l’algorithme AC3, un algorithme de recherche de connexité. Le filtre est notre fonction qui, à chaque itération, élague les listes des manœuvres compatibles restantes pour chaque avion.

## Implémentation d’un filtre initial

Nous implémentons un filtre initial, qui enlève les manœuvres qui n’ont aucune compatibilité avant même le début de l’exploration de l’arbre. Nous espérons réduire le temps d’exécution en réduisant d’emblée la taille des listes à parcourir.

## Benchmarking

Nous lançons un script de benchmarking : celui-ci permet de lancer notre algorithme, avec ou sans les nouvelles fonctionnalités, sur quelques fichiers à la suite. Ainsi, on obtient des données statistiques fiables, grâce au grand nombre de situations couvertes.

Tous les résultats sont présentés dans un tableur présenté en annexe.

## Analyse des résultats

-Les deux dernières fonctionnalités portant sur le filtre n’améliorent pas les performances. On obtient les mêmes résultats avec et sans. En fait, ces fonctionnalités ne réduisent pas le nombre de nœuds explorés, et on remarque que leur incidence sur les temps de calcul est négligeable. L’algorithme AC3 est peut-être mal codé et/ou mal compris.

-Par contre, les résultats sont au rendez-vous pour l’heuristique amélioré. L’algorithme utilisant le nouvel heuristique doit explorer 425 fois moins de nœuds que son prédécesseur plus naïf avant de trouver la solution optimale. De plus, l’écart-type est plus faible, ce qui montre une plus grande polyvalence et efficacité de l’heuristique amélioré.

Le temps de calcul de l’heuristique amélioré est largement compensé par l’économie qu’il apporte à l’algorithme de recherche dans l’arbre. Le nouvel algorithme est plus rapide et parvient donc à traiter plus de situations complexes (c’est pour cela qu’il y a plus de situations répertoriées dans les benchmarks avec l’heuristique amélioré).

# Conclusions

En optimisation, l’utilisation de certaines possibilités algorithmiques se vérifient par des tests. Même si certaines données ne seront pas traitées de manière optimale par un nouvel algorithme, si ce dernier est meilleur dans 90% des cas, on le garde. D’où les études statistiques pour déterminer si on garde, ou non, un bout de code.

Afin d’améliorer encore notre algorithme, on pourrait :

-Purifier le code afin de réduire les temps de calcul de chaque opération

-Tester de nouveaux heuristiques

-Coder une fonction intelligente qui pourrait prévoir le temps de calcul et choisir entre plusieurs heuristiques en fonction du nombre d’avions.

Grâce à ce projet, nous avons en tout cas appris concrètement les enjeux de l’optimisation et de l’intelligence artificielle.

# Annexe : données benchmarking



Ci-dessus : résultats avec filtre initial, avec AC3, comparaison des 2 heuristiques.

Ci-dessous : résultats avec filtre initial, sans AC3, comparaison des 2 heuristiques.

Heuristique amélioré à gauche, naïf à droite.



# Bibliographie

<https://fr.wikipedia.org/wiki/S%C3%A9paration_et_%C3%A9valuation>

<https://ocaml.org/index.fr.html>