# Cours de Machine Learning

RAPPORT DES PARTIES PRATIQUES

Marlier Maxime
ISIB | MAI 2015

# PARTIE 1 : RECHERCHE BASIQUE, HEURISTIQUE ET OPTIMALE

## DESCRIPTION DES ALGORITHMES

### Deph First Search

C'est algorithmes de parcours de graphe fait partie des méthodes de recherche dites aveugles. Il examine une possibilité le plus profondément possible, jusqu'à obtenir un résultat ou un blocage dans quel cas il fait demi-tour pour explorer une autre voie.

Cet algorithme nous assure un résultat (pour peu qu'il existe), mais ne tient pas compte des coûts ni des heuristiques et ne garantit donc aucune optimisation sur le chemin obtenu.

Dans le pire des cas, l'algorithme est obligé de parcourir l'entièreté du graph avant de trouvé une solution

### Non Deterministic Search

Cet algorithme, comme le précédent, découvre le graph de manière aveugle jusqu'à trouver le goal recherché. La seule différence avec DFS est l'introduction d'un caractère aléatoire dans le choix du prochain segment à parcourir dans le graph. De cette manière, l'algorithme ne se focalise pas sur la première branche trouvée. Il est donc statistiquement très peu probable de se trouver dans le « worst case scenario » du DFS. Cet algorithme-ci sera en moyenne plus efficace que DFS.

### Greedy Search

Avec Greedy Search, arrive une notion supplémentaire, celle de l'heuristique d'un nœud par rapport au goal. Chaque nœud sera donc évalué numériquement grâce à une fonction heuristique[1]. Le parcours du graph ne se fera plus de manière complètement aveugle, mais en choisissant des directions vers les nœuds possédant la meilleure valeur heuristique.

Cet algorithme obtiendra de meilleurs résultats que les précédents, en termes de rapidité et d'efficacité, mais ne garantit pas non plus le chemin optimum.

### Estimate-extended Uniform Cost

Ce dernier algorithme est plus complexe à mettre en œuvre, mais a pour avantage de garantir une optimisation du chemin à parcourir. Celui-ci tient compte du coût cumulé des chemins en plus de l'heuristique des nœuds visités.

[1] Fonction heuristique à déterminer d'après l'application choisie

Cependant il se peut que le chemin obtenu ne soit pas le meilleur dans tous les cas. (Si le dernier saut vers le goal possède un coût bien plus élevé que la moyenne par exemple.)

### Branch-and-Bound

Cette dernière technique va parfaire nos algorithmes de recherche en incluant la notion de coût accumulé. Cela permettra de vérifier que parmi tous les chemins trouvés vers le goal, nous ne gardons bien que celui au coût total le plus faible.

## APPLICATION

L'exemple choisi pour l'implémentation des différents algorithmes est une recherche de chemin sur une carte. La carte est représentée sous forme de graphe, où les sommets représentent des villes et les arêtes sont les chemins les reliant. Par mesure de facilité d'implémentation, l'exemple présenté est fictif mais représentatif d'une situation réelle[2].

## CHOIX DES COÛTS ET HEURISTIQUES

L'heuristique attribuée à un sommet est la distance à vol d'oiseau jusqu'au goal sélectionné. Le coût attribué à un chemin, et sa longueur réelle.

Ces deux mesures sont exprimées en pixels et sont calculées automatiquement à l'initialisation. Cela permet un dynamisme complet du graph en termes de choix de départ et d'arrivée, et une automatisation de test sur les algorithmes, en faisant varier ces deux points.

## OUTILS DE DÉVELOPPEMENT

- Le langage utilisé pour la programmation des algorithmes est le c++.
- Les parties graphiques sont réalisées avec les librairies OpenFrameworks.

[2] Il peut d'ailleurs être adapté à une situation réelle en remplaçant notre graphe par la représentation d'une vraie carte, ce qui demandait beaucoup de travaille pour obtenir une représentation juste de la réalité, et ce n'est pas le but de l'exercice, mais bien l'utilisation et la comparaison des algorithmes de recherche.

# COMPARAISON DES RÉSULTATS

Afin de réalisé une comparaison significative, Les 4 algorithmes ont été testé de manière automatique en envisageant toutes les possibilités de START et GOAL parmi les nœuds du graph. (6560 possibilités en tout).

Les résultats complets de ce test se trouvent dans le fichier Excel joint à ce rapport.

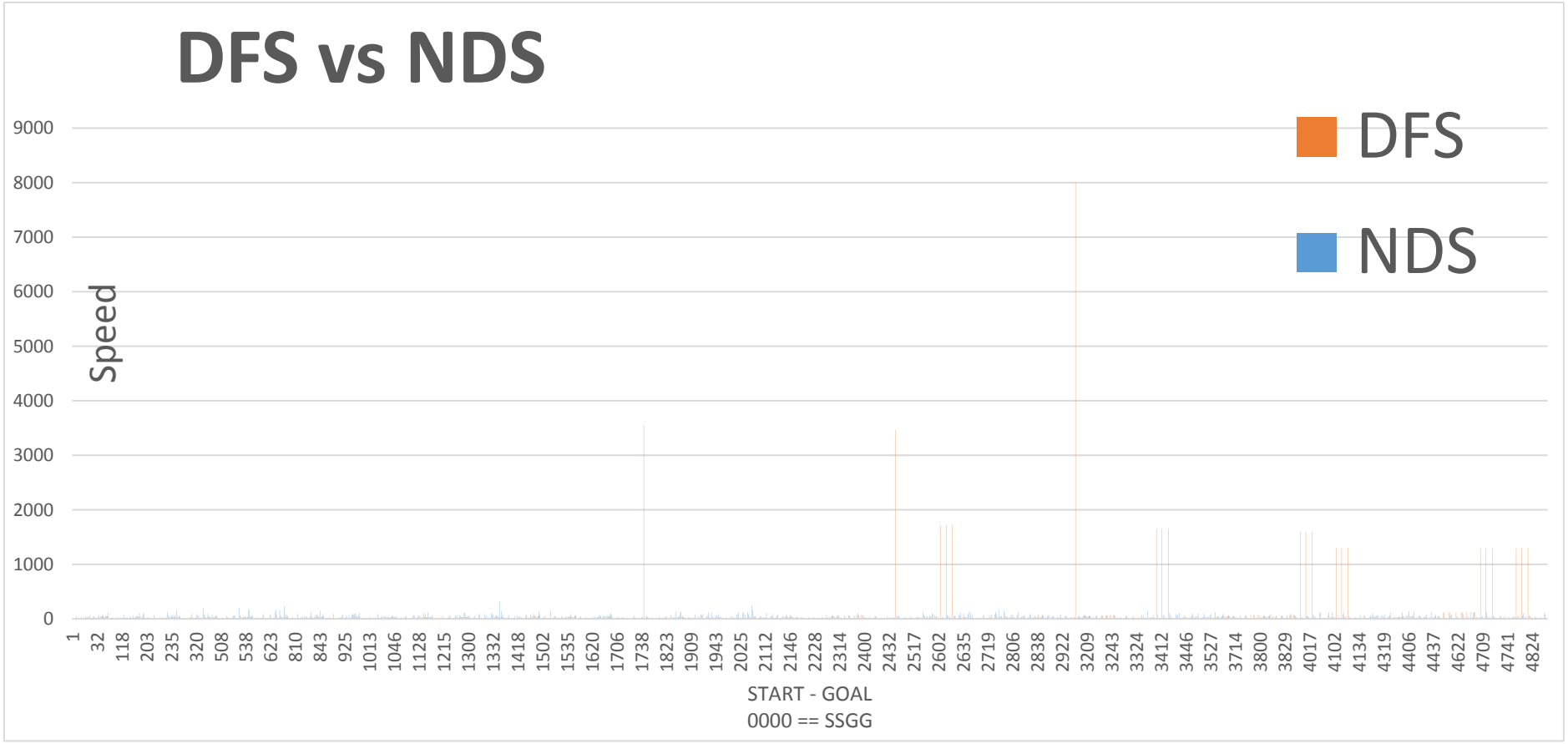
Voici quelques comparaisons significatives :

(page suivante)

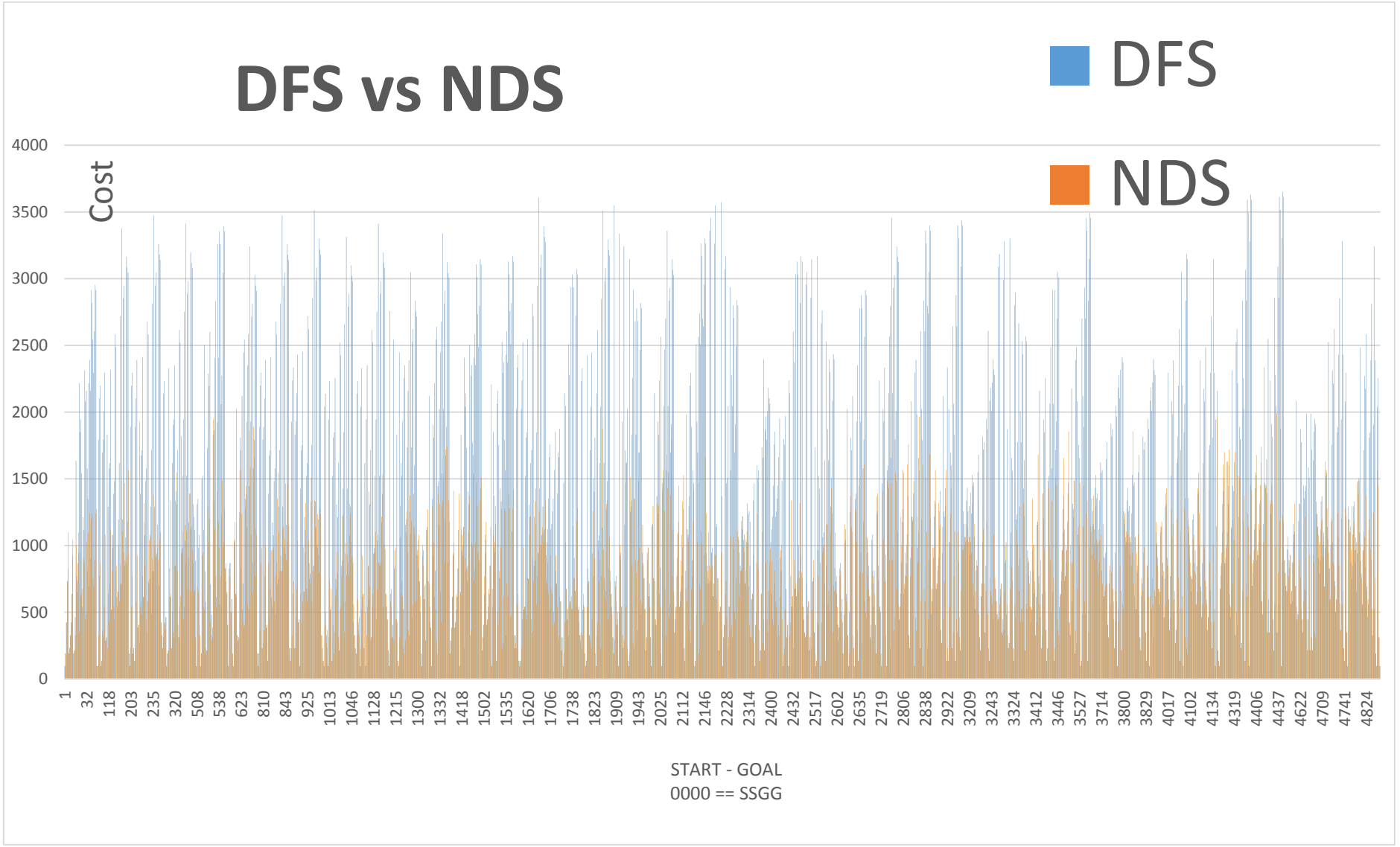
# DFS vs NDS

Cette comparaison nous permet d'observer les avantages d'une introduction d'élément aléatoire dans notre recherche aveugle.

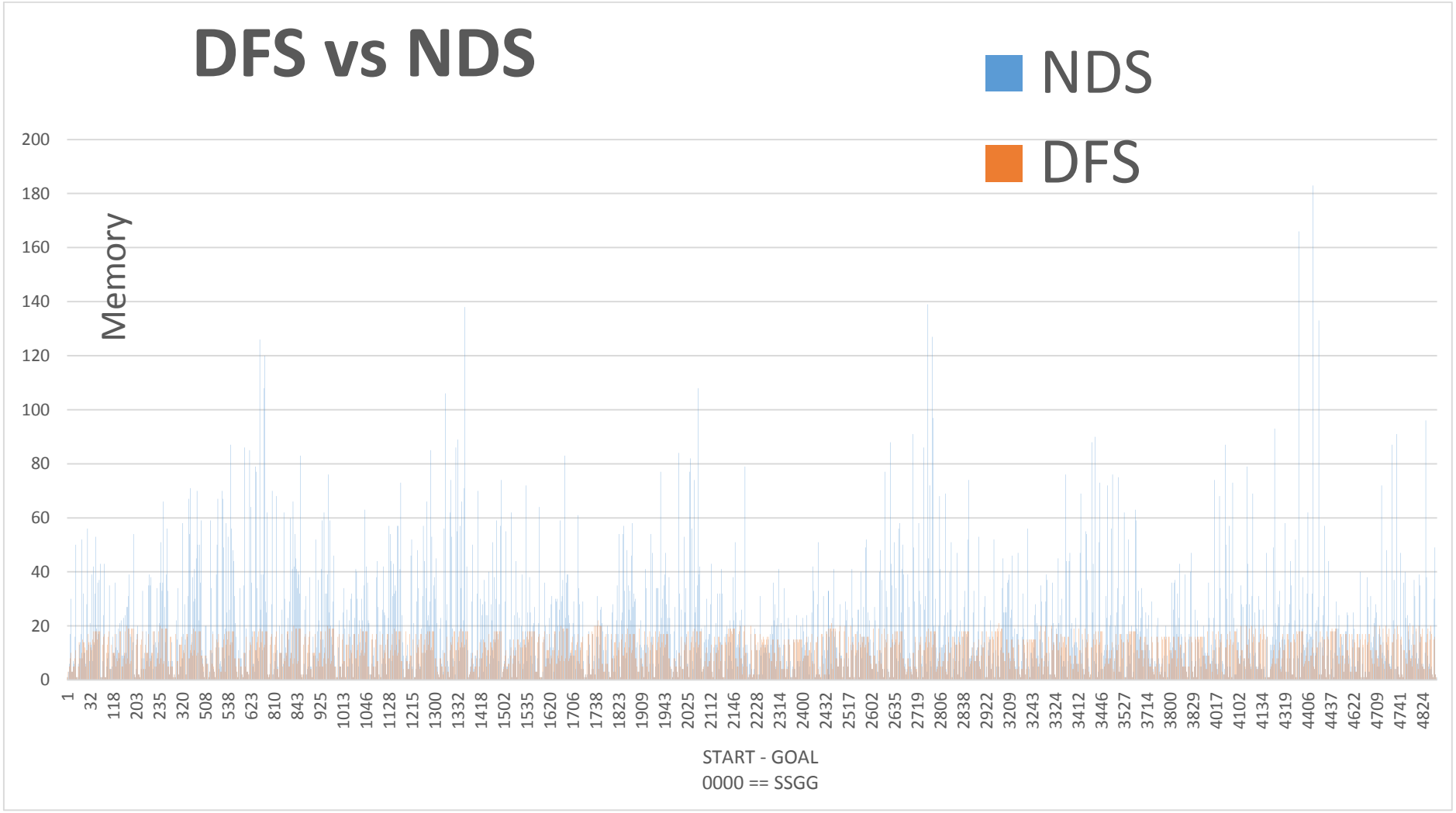
En termes de vitesse par exemple (exprimée comme le nombre d'étapes effectuées par l'algorithme) Nous observons qu'en moyenne, DFS et NDS sont équivalents. Mais Dans certains cas (les pires cas) DFS continuant à chercher une solution par la première voie empruntée, se trouve dans une boucle très longue lui demandant un nombre d'étapes très important (jusqu'à 100 fois plus grand)



En termes de coût (exprimé par le coût accumulé du chemin final sélectionné) L'introduction de l'aléatoire est une nouvelle fois bénéfique.



Pas contre NDS nécessite souvent plus de mémoire, car il envisage plus de possibilités différentes que DFS.



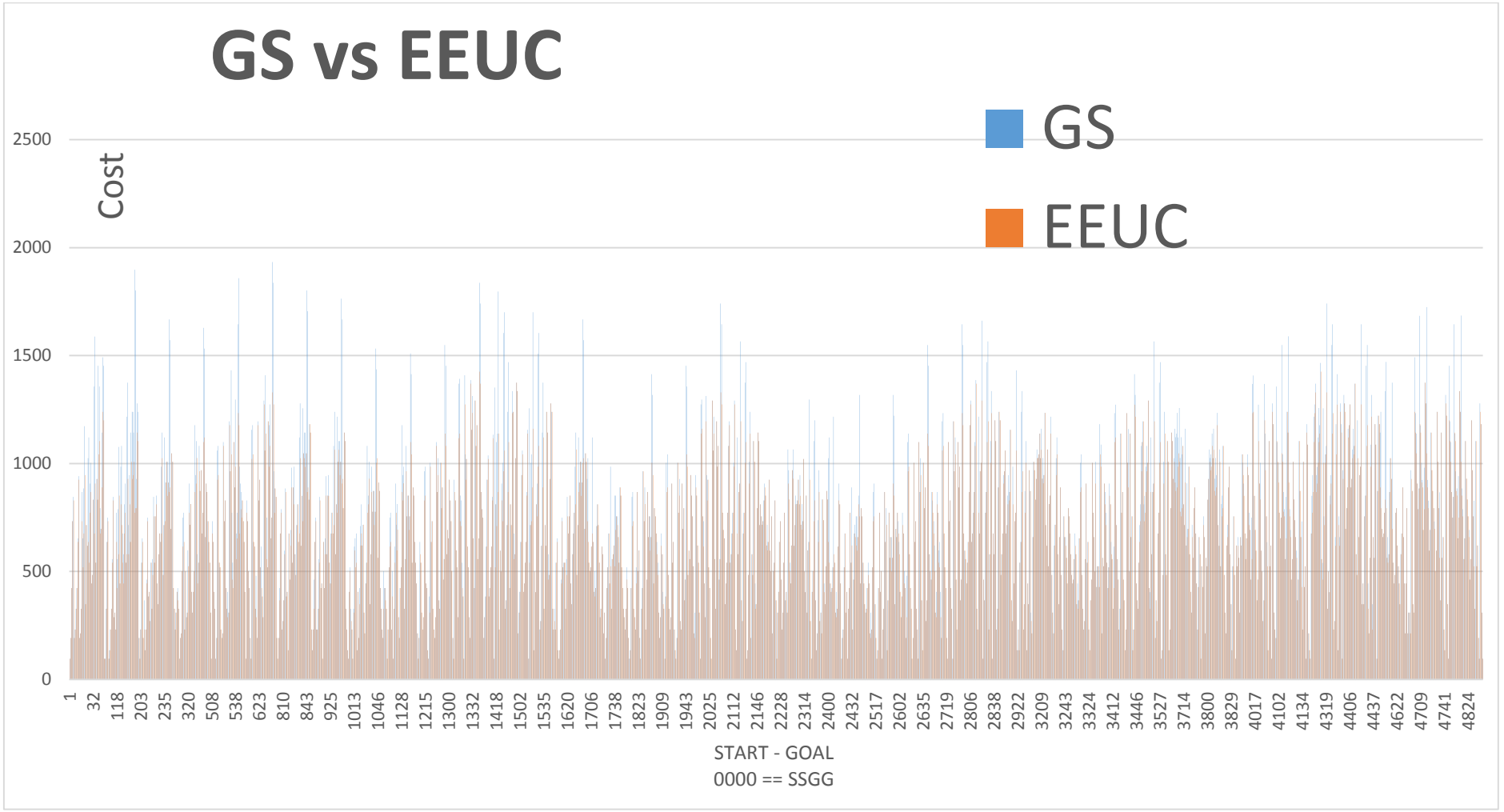
# CONCLUSIONS

L’algorithme NDS offre des biens meilleures performances que DFS en termes de rapidité et de cout. Il demande néanmoins des ressources mémoire plus importantes.

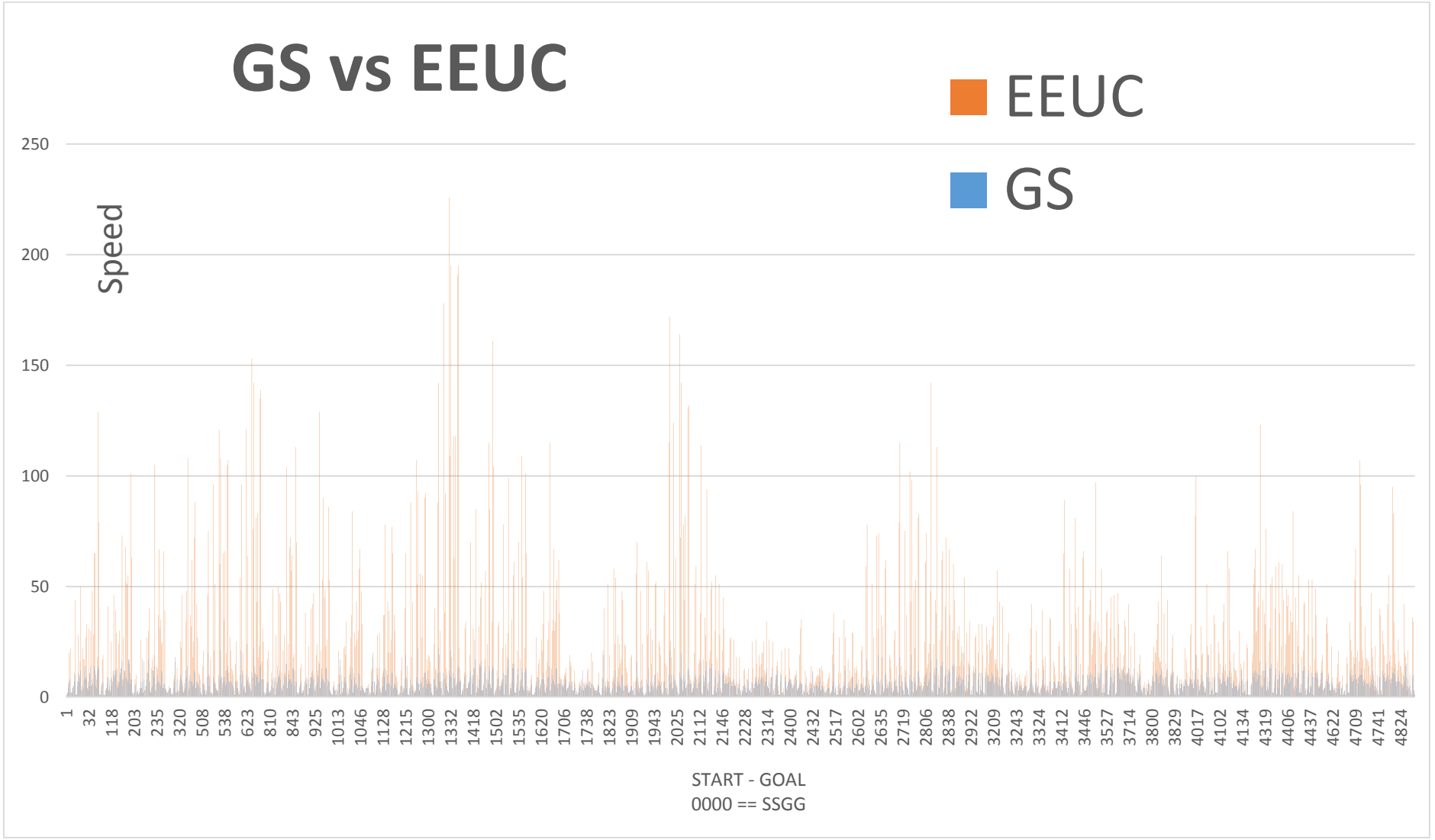
# Gready Search vs Estimate-extended Uniform Cost

Voilà deux algorithmes tenant compte des notions d'heuristiques, le deuxième utilisant en plus le coût cumulé pour trier les chemins.

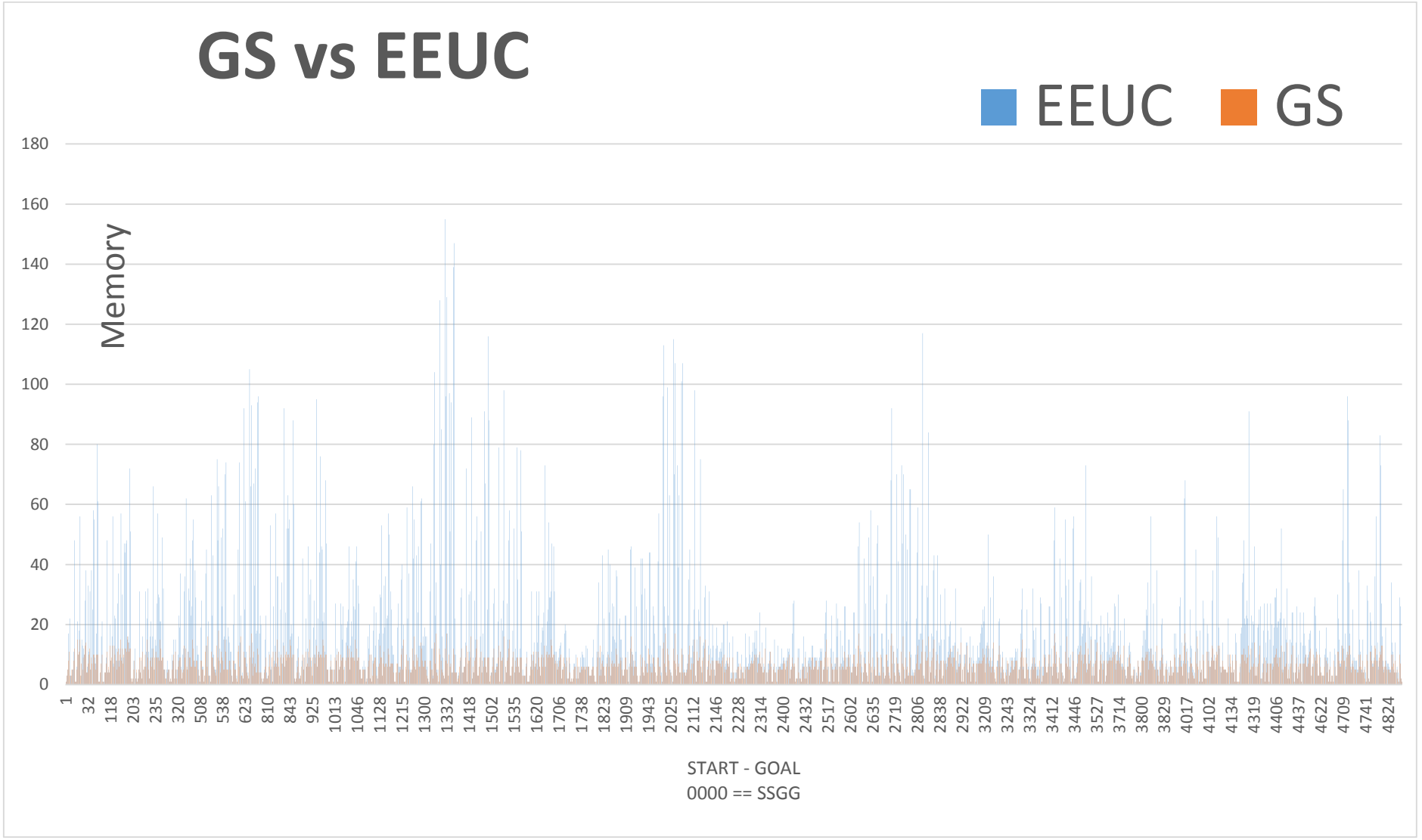
Nous voyons ici que les résultats de Gready Search en termes de coût sont légèrement inférieurs à EEUC.



En termes de vitesse maintenant, nous constatons que Gready Search est bien plus performant.



En ce qui concerne la mémoire, Gready Search est aussi bien moins gourmand !



# CONCLUSIONS

Estimate-extended Uniform Cost (avec branched bound) nous donne la certitude de trouver le chemin au coût le plus faible, mais au prix d'une vitesse de calcul bien inférieure et des besoins en mémoire bien plus grand que Gready Search. Ces pourquoi ce dernier est un bon compromis car ses résultats en termes de chemin trouvé sont assez satisfaisants.

# PARTIE 2 : ALGORITHMES DE JEUX DE STRATÉGIE POUR DEUX :

# LE JEUX DU MOULIN

## DESCRIPTION DU JEU

Voilà la description du jeu mis en œuvre (description reprise de la page Wikipedia du Jeu du Moulin).

### Matériel

- Un tablier formé de trois carrés imbriqués offrant vingt-quatre intersections
- neuf pions blancs
- neuf pions noirs

### Règles du jeu

Le jeu se déroule en deux temps : la pose puis le mouvement.

- **La pose** : tant qu'il en possède encore, chaque joueur place à tour de rôle un pion sur une intersection libre.
- **Le mouvement** : lorsqu'il n'a plus de pion à poser, chaque joueur fait glisser l'un de ses pions vers une intersection voisine libre en suivant un chemin prévu.

À tout moment du jeu, celui qui réalise un moulin, c'est-à-dire l'alignement de trois de ses pions, peut capturer un pion adverse quelconque parmi ceux n'appartenant pas à un moulin.

Le jeu s'achève quand un joueur n'a plus que deux pions ou ne peut plus jouer, il est alors le perdant.

## OUTILS DE DÉVELOPPEMENT:

Le choix des outils de développement est le même que pour l'exercice précèdent.

# DESCRIPTION DES ALGORITHMES

## Min-Max

MinMax est un algorithme cherchant à minimiser les pertes maximum, c'est-à-dire minimiser le pire des cas. Dans le domaine des jeux à deux joueur à somme nulle (ou information complète), il amène l'ordinateur à calculer toutes les possibilités de coups jusqu'à une certaine profondeur limite (nombre de coups d'avance) et d'évaluer de manière statique les grilles obtenues afin de pouvoir maximiser ces propre coups et minimiser ceux de l'adversaire.

## Alpha-beta

Cette technique permet de réduire le nombre de nœuds calculé par l'algorithme MinMax. Cela permet de réduire les temps de calcul et donc d'atteindre une profondeur plus importante et ainsi améliorer les performances de l'IA.

# FONCTION D'ÉVALUATION

Les moyens de gagner une partie de Jeu du Moulin sont les suivants :

- Eliminer 7 pions de l'adversaire
- Ou Bloquer la totalité de ses pions restants

Il y a donc plusieurs critères à prendre en comptes pour évaluer une grille :

1. Le nombre de moulins établis pas les deux jours
2. Le nombre de « presque moulins »
3. Le nombre de pion encore en jeux
4. Le nombre de possibilités restantes

Ce sont 4 variables à comptabiliser pour chaque joueur.

*Note : un moulin est l'alignement consécutif de 3 pions alors qu'un « presque moulin » est l'alignement de 2 pions **AVEC** le troisième emplacement disponible.*

Ce qui représente l'évaluation d'une grille est la somme des différences entre les joueurs concernant chaque variable.

Une pondération peut encore être ajoutée à chaque variable, car celles-ci n'ont pas forcément le même poids.

Un moulin complet a par exemple plus d'importance qu'un « presque moulin » pour un joueur.

Si nous avons P1 et P2 représentant les joueurs (p2 étant l'IA qui veux maximiser) et respectivement v1, v2, v3 et v4 étant les variables d'évaluation décrites ci-dessus, alors :

EVAL = R1*(p1->v1 – p2->v1) + R2*(p1->v2 – p2->v2) + R3*(p1->v4 – p2->v4) + R4*(p1->v4 – p2->v4)

R1, R2, R3 et R4 étant les pondération de chaque variable.

# COUPS D'AVANCE (PROFONDEUR)

La profondeur maximale de 5 est atteinte sans optimisation alpha-beta. Avec un temps de latence très raisonnable (1 à 2 secondes) pour les arbres les plus grands.

Une profondeur de 6 implique un temps de calcul beaucoup trop long pour être jouable. Ceci étant, les résultats sont déjà très satisfaisants, et l'algorithme est difficile à battre.

L'optimisation alpha-beta nous permet de descendre à une profondeur de 8. Ce qui signifie 4 coups par joueur et représente un niveau très bon.

Certaines optimisations sont encore possible au niveau des structures de données, afin d'optimiser le temps de calcul pour la génération des enfants à chaque niveau. Cependant, il sera difficile de descendre plus profondément dans l'algorithme sachant que la génération des niveaux dans l'arbre ressemble à une exponentiel. Un avantage que présente le jeu du moulin est qu'au fur et à mesure des mouvements dans le jeu, le nombre de coups possibles est considérablement réduit. On peut donc imaginer descendre plus profondément en fonction du nombre de coups possible au tour suivant.

# PARTIE 3 : PROGRAMMATION PAR CONTRAINTES

## LANGAGE ET LIBRAIRIES

Pour cette partie pratique, le langage utilisé sera Python. Le module *python-constraint* nous permettra de modéliser des problèmes sous forme de variables appartenant à des domaines de valeurs et aussi imposer des contraintes sur ces variables afin de résoudre les problèmes.

## APPLICATIONS

Parmi les énoncés proposés, nous allons en résoudre trois. Le problème du zebre, et deux exemples de crypto-arithmétique (SEND + MORE = MONEY && DONALD + GERALD = ROBERT).

Les énoncés sont tirés d'un recueil d'exercices.

## LE PROBLEME DU ZEBRE

### Énoncé

On s'intéresse au problème suivant :

Cinq maisons consécutives, de couleurs différentes, sont habitées par des hommes de différentes nationalités. Chacun possède un animal différent, a une boisson préférée différente et fume des cigarettes différentes. De plus, on sait que :

- Le norvégien habite la première maison,
- La maison à côté de celle du norvégien est bleue,
- L'habitant de la troisième maison boit du lait,
- L'anglais habite la maison rouge,
- L'habitant de la maison verte boit du café,
- L'habitant de la maison jaune fume des kools,
- La maison blanche se trouve juste après la verte,
- L'espagnol a un chien,
- L'ukrainien boit du thé,
- Le japonais fume des cravens,
- Le fumeur de old golds a un escargot,
- Le fumeur de gitanes boit du vin,
- Le voisin du fumeur de Chesterfields a un renard,
- Le voisin du fumeur de kools a un cheval.

Qui boit de l'eau ? A qui appartient le zèbre ?

## MODÉLISATION DU PROBLÈME

Nous pouvons déterminer le domaine de valeur comme les numéros de maison (de 1 à 5) et les variables comme les caractéristiques de ces maisons (nationalité, marque de cigarette, animal de compagnie, couleur des murs et boisson préférée !)

```
prob = Problem()
prob.addVariables([
 "blanche",
 "rouge",
 "verte",
 "jaune",
 "bleue",
 "norvegien",
 "anglais",
 "ukrainien",
 "japonais",
 "espagnol",
 "cheval",
 "renard",
 "zebre",
 "escargot",
 "chien",
 "the",
 "eau",
 "lait",
 "cafe",
 "vin",
 "kools",
 "chesterfields",
 "old_golds",
 "cravens",
 "gitanes"], [1, 2, 3, 4, 5])
```

Il faut ensuite définir une contrainte pour chacune des assertions de l'énoncé.

- norvégien = 1,
- bleue = norvégien + 1,
- lait = 3,
- anglais = rouge,
- verte = café,
- jaune = kools,
- blanche = verte + 1,
- espagnol = chien,
- ukrainien = thé,
- japonais = cravens,
- old_golds = escargot,
- gitanes = vin,
- (chesterfields = renard + 1) ou (chesterfields = renard - 1),
- (kools = cheval + 1) ou (kools = cheval - 1)

```
prob.addConstraint(lambda norvegien: norvegien == 1, ["norvegien"])
prob.addConstraint(lambda bleue, norvegien: bleue == norvegien + 1 , ["bleue","norvegien"])
prob.addConstraint(lambda lait: lait == 3, ["lait"])
prob.addConstraint(lambda anglais, rouge: anglais == rouge, ["anglais", "rouge"])
prob.addConstraint(lambda verte, cafe: verte == cafe, ["verte", "cafe"])
prob.addConstraint(lambda jaune, kools: jaune == kools, ["jaune", "kools"])
prob.addConstraint(lambda blanche, verte: blanche == verte + 1, ["blanche", "verte"])
prob.addConstraint(lambda espagnol, chien: espagnol == chien, ["espagnol", "chien"])
prob.addConstraint(lambda ukrainien, the: ukrainien == the, ["ukrainien", "the"])
prob.addConstraint(lambda japonais, cravens: japonais == cravens, ["japonais", "cravens"])
prob.addConstraint(lambda old_golds, escargot: old_golds == escargot, ["old_golds", "escargo
t"])
prob.addConstraint(lambda gitanes, vin: gitanes == vin, ["gitanes", "vin"])
prob.addConstraint(lambda chesterfields, renard: chesterfields == renard - 1 or chesterfield
s == renard + 1, ["chesterfields", "renard"])
prob.addConstraint(lambda kools, cheval: kools == cheval - 1 or kools == cheval + 1, ["kools
", "cheval"])
```

Il faut également ajouter des contraintes pour spécifier qu'on ne peut avoir plusieurs maisons de la même couleur, ni plusieurs maisons avec le même animal, ...

```
prob.addConstraint(AllDifferentConstraint(), ["blanche", "rouge", "verte", "jaune", "bleue"]
)
prob.addConstraint(AllDifferentConstraint(), ["the", "eau", "lait", "cafe", "vin"])
prob.addConstraint(AllDifferentConstraint(), ["norvegien", "anglais", "ukrainien", "japonais
", "espagnol"])
prob.addConstraint(AllDifferentConstraint(), ["cheval", "renard", "zebre", "escargot", "chie
n"])
prob.addConstraint(AllDifferentConstraint(), ["kools", "chesterfields", "old_golds", "craven
s", "gitanes"])
```

## SOLUTION(S)

Le problème posé comme tel possède une solution unique que voici :

La maison jaune est habitée par un renard et un norvégien qui boit de l'eau et fume des kools

La maison bleue est habitée par un cheval et un ukrainien qui boit du thé et fume des chesterfields

La maison rouge est habitée par un escargot et un anglais qui boit du lait et fume des old_golds

La maison verte est habitée par un zèbre et un japonais qui boit du café et fume des cravens

La maison blanche est habitée par un chien et un espagnol qui boit du vin et fume des gitanes

Nous apprenons ainsi que c'est le norvégien qui boit de l'eau et que le zèbre habite avec le Japonais dans la maison verte!

# SEND MORE MONEY

## Énoncé

On considère l'addition suivante :

```
   SEND
 + MORE
-------
= MONEY
```

Où chaque lettre représente un chiffre différent (compris entre 0 et 9). On souhaite connaitre la valeur de chaque lettre, sachant que la première lettre de chaque mot représente un chiffre différent de 0.

### MODÉLISATION DU PROBLÈME

Ici, les inconnues sont les valeurs des lettres S, E, N, D, M, O, R et Y. On a donc 8 variables, chacune pouvant prendre une valeur comprise entre 0 et 9, sauf S et M qui ne peuvent être égales à 0 puisqu'elles sont en début de mot.

En termes de contraintes, il faut d'abord préciser que chaque lettre est différente.

Ensuite concernant la définition de l'équation à résoudre, il existe deux manières d'aborder de problème.

Une première possibilité consiste à définir une seule contrainte "traduisant" l'équation SEND + MORE = MONEY. Il faut donc reconstituer le nombre associé à chaque mot à partir des valeurs des lettres le composant (par exemple, le nombre associé au mot SEND est égal à D + 10N + 100E + 1000S).

Une autre possibilité, pour définir les contraintes, consiste à poser les contraintes "verticalement", comme quand on fait une addition à la main :

La somme des chiffres des unités des nombres à additionner est égale au chiffre des unités du nombre résultat, plus dix fois la retenue ;

La somme de la retenue des unités et des chiffres des dizaines des nombres à additionner est égale au chiffre des dizaines du nombre résultat, plus dix fois la retenue : etc...

Avec une telle modélisation, il faut rajouter 3 variables R1, R2 et R3 correspondant aux retenues successives, et pouvant prendre pour valeur 0 ou 1 (car la somme de deux chiffres compris entre 0 et 9 est toujours inférieure ou égale à 18).

### MODÉLISATION DE LA SOMME :

Pour cet exercice, nous choisirons la première modélisation qui consiste à l'écriture de l'équation sous forme d'une seule contrainte que voici :

1000*S + 100*E + 10*N + D +1000*M + 100*O + 10*R + E == 10000*M + 1000*O + 100*N + 10*E + Y

# Solution(s)

Le problème ainsi posé nous offre une solution unique :

```
   SEND      9567
 + MORE    + 1085
-------    ------
= MONEY     10652
```

# Résultats et comparaison des algorithmes

La librairie Python constraint propose trois algorithmes pour la résolution des problèmes.

- MinConflictsSolver
- BacktrackingSolver
- RecursiveBacktrackingSolver

Voici une comparaison de ces algorithmes sur SEND + MORE = MONEY

```
$ python MinConflictsSolver.py
temps de calcul = 0.499 secondes
Traceback (most recent call last):
  File "MinConflictsSolver.py", line 48, in <module>
    print "   SEND \t    %d%d%d%d" % (sol["S"], sol["E"], sol["N"],
sol["D"])
TypeError: 'NoneType' object has no attribute '__getitem__'

$ python BacktrackingSolver.py
temps de calcul = 6.720 secondes
   SEND           9567
 + MORE         + 1085
-------         -------
= MONEY         = 10652

$ python RecursiveBacktrackingSolver.py
temps de calcul = 0.118 secondes
   SEND           9567
 + MORE         + 1085
-------         -------
= MONEY         = 10652
```

Nous pouvons observer que sur ces trois algorithmes, seulement 2 fournisse une solution au problème. Et que l'un d'entre eux offre de bien meilleurs résultats !

Gagnant des algo = RecursiveBacktrackingSolver !

Il offre la solution en un temps de calcul nettement meilleur que BacktrackingSolver :

`0.118 secondes` contre `6.720 secondes`

# DONALD + GERALD = ROBERT?

Voilà le même problème avec une équation différente.

## Solution

```
  DONALD          526485
+ GERALD        + 197485
--------        --------
= ROBERT        = 723970
```

## Optimisations

La modélisation est identique à l'énoncé précédent, cependant nous pouvons introduire ici quelques optimisations qui nous feront gagner un temps de calcul considérable.

En observant ce problème de plus proche nous constatons que :

- Le résultat de cette somme possède le même nombre de digit que ces facteurs.
  Cela signifie que la somme des lettres D et G doit être strictement plus petite que 10 (sans quoi il y aurait un report et donc un niveau supplémentaire dans le résultat)

$$D + G < 10$$

- La dernière lettre du résultat ('T') est la somme de 2 lettres identiques ('G'). Cela signifie que la variable 'T' est paire !

$$T \% 2 == 0$$

# RÉSULTATS

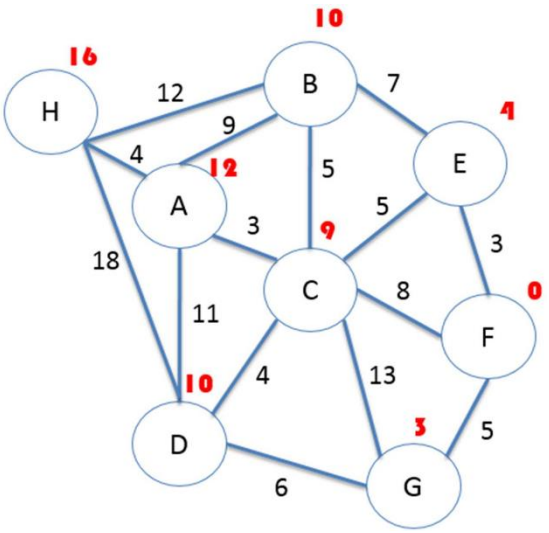
Grâces à ces optimisations le temps de calcul est divisé par 10 ! Nous passons de 50 secondes environs à 5 secondes pour trouver la solution. Cela montre qu'il est nécessaire d'observer attentivement un problème de programmation par contraintes au-delà des informations explicites imposées afin de réduire les calculs nécessaires et d'obtenir un résultat plus rapide.

# PARTIE 4 : EXERCICES ANNEXES

## EXERCICE 1 – ALGORITHMES DE RECHERCHE

Voici le problème de réseau téléphonique suivant : Un message doit transiter de la Hongrie (H) à la Finlande (F), par le chemin le plus rapide possible.

Les poids sur les arcs représentent les durées de transmission en milliseconde entre les différents pays. Les poids sur les nœuds représentent l'heuristique attribuée à chaque nœud du réseau.




1. **Utiliser l'algorithme *Bi-directional Search* sur ce réseau, pour trouver le chemin le plus rapide. Décrire l'état de la file à chaque étape et montrer dans l'arbre de recherche quel est le chemin emprunté de H à F.**

**Bi-directional Search** exécute de manière parallèle n'importe quel algorithme de recherche. Il maintient des files contenant les chemins trouvés, l'une depuis le point de départ et l'autre depuis l'arrivée. Dès que les deux files se rencontrent en un nœud**, BDS** a trouvé une solution.

Voici une utilisation de BFS avec BDS :

Q1 : [H]
Q2 : [F]

Q1 : [HA, HB, HD]
Q2 : [FC, FE, FG]

Q1 : [HB, HD, HAB, HAC, HAD]
Q2 : [FE, FG, FCA, FCB, FCD, FCE]

Le chemin ainsi obtenu est [H-A-C-F]

2. **Quelle est la taille de ce chemin en millisecondes ?**

[H-A-C-F] dure 15 millisecondes

3. **En combien d'étapes arrive-t-on à la solution ?**

3 étapes sont nécessaires à BDS pour trouver une solution

4. **Maintenant, si chaque pays dans le réseau prend 2 euros par message téléphonique qui passe sur son territoire, et que votre unique objectif est d'optimiser le coût de vos messages téléphoniques transitant entre la Hongrie et la Finlande, plus la rapidité de transmission, quel algorithme de recherche devriez-vous utiliser pour atteindre cet objectif?**

L'exemple ci-dessus obtient déjà le chemin le moins coûteux. En effet l'utilisation de BFS nous garantit le chemin le plus court (en nombre de sauts).

BFS est donc une bonne solution.

5. **Quel serait alors le meilleur chemin à prendre ?**

Ici 3 réponses sont possibles car trois chemins sont équivalents en termes de pays traversé :

- [H-A-C-F]
- [H-D-G-F]
- [H-B-E-F]

# EXERCICE 2 – LOGIQUE FLOUE

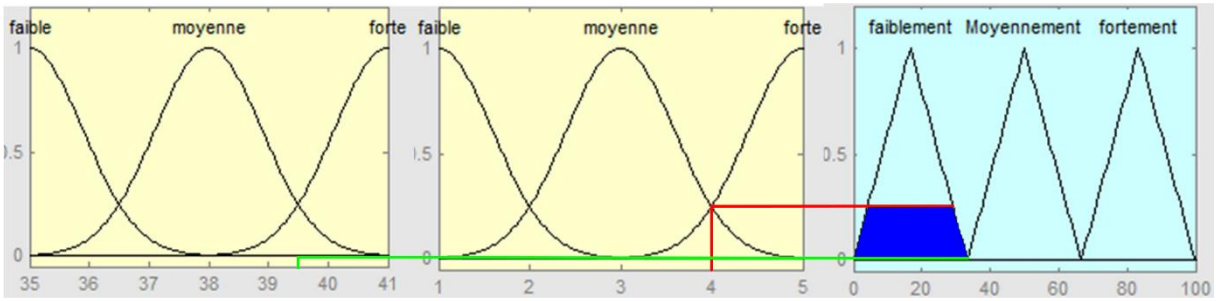
Vous devez apprendre à une machine à conduire en agglomération selon vos règles. Pour cela, on considère les variables floues décrites graphiquement ci-dessous :

- 2 entrées : la vitesse (de 35 à 41 km/h) et la distance par rapport à un passage pour piéton (de 1 à 5, en centaines de mètres où 1 = 100m et 5 = 500m).
- 1 sortie : pourcentage de force sur la pédale de frein (de 0 à 100 % où 0 = 0 Newtons et 100% = 50 Newtons).

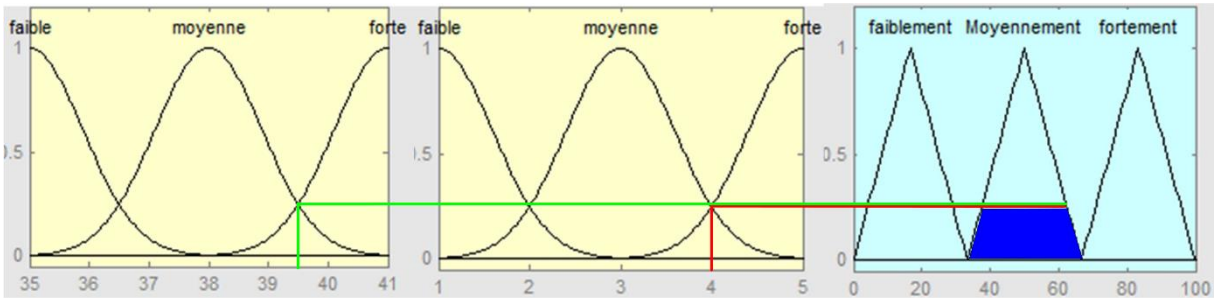
D'après vous, en vous basant sur les variables floues représentées ci-dessous, si la vitesse de votre véhicule est de

39.5 Km/h  et que la distance par rapport au passage pour piéton est de 400m, quelle sera la force à appliquer sur La pédale de frein en Newtons ? Voici les règles utilisées :

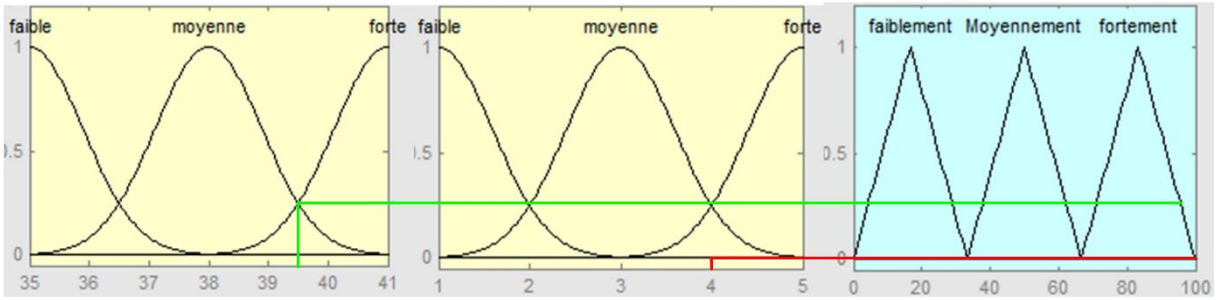
1. Si votre vitesse est faible ou que votre distance est forte, il faudra appuyer faiblement.



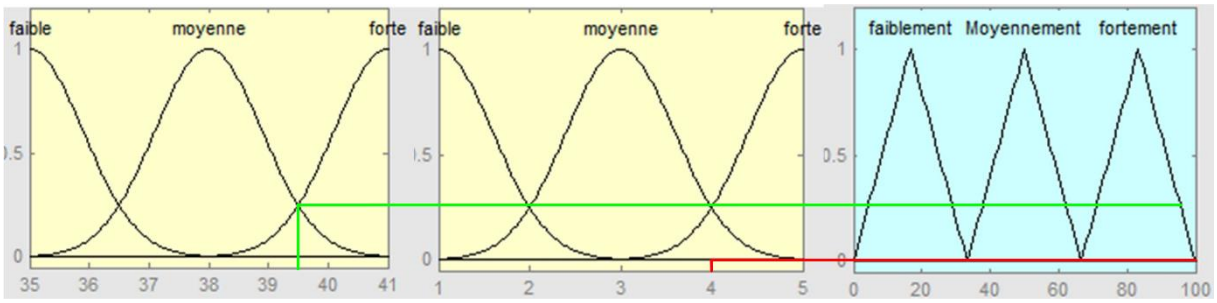

2. Si votre vitesse est moyenne et que votre distance est moyenne, il faudra appuyer moyennement.



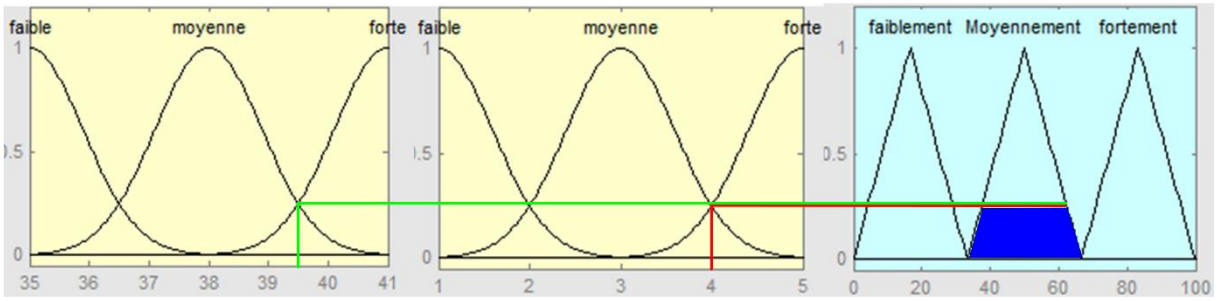

3. Si votre vitesse est moyenne est que votre distance est faible, il faudra appuyer fortement.




4. Si votre vitesse est forte et que votre distance est faible, il faudra appuyer fortement.

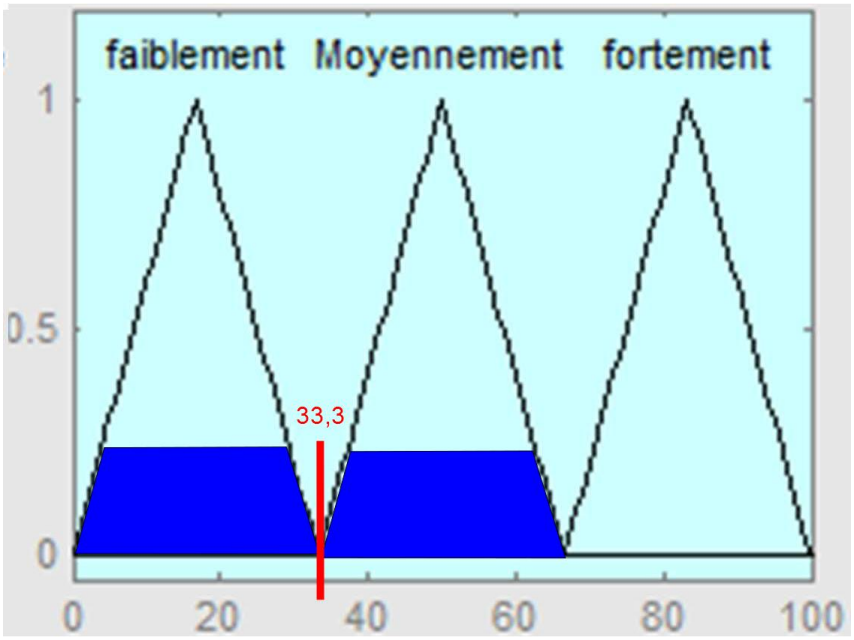



5. Si votre vitesse est forte et que votre distance est forte, il faudra appuyer moyennement.




# Résultat

La force appliquée sur la pédale de frein est donc de 33,3 Newton d'après les règles de logique floue décrites ci-dessus.



# BIBLIOGRAPHIE


- OpenFrameworks : http://openframeworks.cc/
- python-constraint : http://labix.org/python-constraint
- Règles du jeu du moulin : http://fr.wikipedia.org/wiki/Jeu_du_moulin
- Énoncés des problèmes de programmation par cntrainte : http://liris.cnrs.fr/csolnon/Site-PPC/session2/e-miage-ppc-sess2.htm