Sorbonne Université

Modèles et algorithmes pour la décision multicritère et collective

MADMC : Projet

Auteurs :
BIEGAS Emilie (3700036)
Kadhi Youssef (3680916)

janvier 2022

Table des matières

1 Première procédure de résolution

1.1 Procédure Implémentée

1.1.1 Phase 1 - PLS

PLS (listes classiques)

Dans cette première procédure, pour déterminer la solution préférée d'un décideur il faut tout d'abord avoir une approximation des points non dominés au sens de Pareto. Pour ce faire nous avons, grâce aux fonctions suivantes, implémenté la méthode de recherche locale de Pareto avec n objectifs.

```
popInitiale(W,w):
    """Fonction initialisant la population
    La population initiale sera constituée d'une seule solution réalisable, générée aléatoirement :
    tant qu'il reste de la place dans le sac, un objet choisi aléatoirement (de poids inférieur a la quantité de
     place restante) est ajoute
    Paramètres : W est la capacite du sac
                 w est la liste de poids des objets
    Retourne une population initiale sous la fome de liste de liste (liste contenant une seule solution)"""
```

```
Voisin(x, W, w):
    """Fonction permettant de generer les voisins d'une solution
    Paramètres : x est le vecteur dans lequel x[i] indique 1 si on a pris l'objet i et 0 sinon
                 W est la capacite du sac
                 w est la liste de poids des objets
    Retourne N la liste des voisins"""
```

```
non_domine(a, b, v):
    """Fonction qui renvoie vrai si a n'est pas domine par b et faux sinon, cad si a est domine par b
    Paramètres : a est un vecteur dans lequel a[i] indique 1 si on a pris l'objet i et 0 sinon (vecteur caracté
     risant une solution)
                 b est un autre vecteur caractérisant une solution, cad dans lequel b[i] indique 1 si on a pris l
     'objet i et 0 sinon
                 v est la matrice de valuation des objets sur les différents critères
    Retourne vrai si a n'est pas domine par b et faux sinon, cad si a est domine par b"""
```

```
MiseAJour(X, x, v):
    """Mise a jour de la liste X en cherchant a ajouter potentiellement x
    Cette procédure renverra un boolean egal a Vrai si la solution a été ajoutée dans la liste
    Paramètres : X un ensemble de solutions (une solution est un vecteur dans lequel sol[i] vaut 1 si on a pris l
     'objet i et 0 sinon)
                 x est une nouvelle solution (un vecteur dans lequel x[i] indique 1 si on a pris l'objet i et 0
     sinon)
                 v est la matrice de valuation des objets sur les différents critères
    Retourne vrai si on a ajoute x a X et faux sinon (cad si x etait domine par un element de X)"""
```

```
PLS(W, w, v):
    """Fonction PLS
    Paramètres : W est la capacite du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les differents critères
    Retourne une approximation de l'ensemble des solutions efficaces"""
```

PLS (ND-Tree)

Pour réduire le temps de calcul de cette approximation des points non dominés sur plusieurs objectifs, nous avons implémenté les fonctions suivantes servant à mémoriser le front sous la forme de ND-Tree.

```
Insertion_ND_Tree(ND_noeuds, newx, newy, nbCriteres):
    """Fonction permettant de gerer les dominances dans le ND-Tree lorsque l'on veut inserer la solution newx de
     vecteur de valuation newy
    Paramètres : ND_noeuds l'ensemble des noeuds du ND-Tree
                 newx la solution que l'on souhaite ajouter au ND-Tree
                 newy le vecteur de valuation de la solution que l'on souhaite ajouter au ND-Tree
                 nbCriteres le nombre de critères a considérer
    Retourne Vrai si le noeud a ete insere et Faux sinon"""
```

```
Voisins_ND_Tree(ND_noeuds, TailleND_noeud, x, W, w, v):
    """Fonction gérant le voisinnage du noeud x (en insérant dans le ND-Tree les voisins si nécessaire)
    Paramètres : ND_noeuds l'ensemble des noeuds du ND-Tree
                 TailleND_noeud le nombre maximum de points dans un noeud du ND-Tree
                 x est le vecteur dans lequel x[i] indique 1 si on a pris l'objet i et 0 sinon
                 W est la capacite du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères"""
```

```
Separation_ND_Tree(ND_noeuds, TailleND_noeud, nbCriteres):
    """Fonction vérifiant que la capacité des noeuds est respectée et permettant de séparer un noeud en deux s'il
      dépasse la
    capacite maximale d'un noeud
    Paramètres : ND_noeuds l'ensemble des noeuds du ND-Tree
                 TailleND_noeud le nombre maximum de points dans un noeud du ND-Tree
                 nbCriteres le nombre de critères à considérer
    """
```

```
nadir_ideal(ND_noeud, nbCriteres):
    """Fonction permettant de calculer le point nadir et le point idéal d'un noeud du ND-Tree
    Paramètres : ND_noeud un noeud du ND-Tree
                 nbCriteres le nombre de critères a considérer
    """
```

```
PLS_ND_Tree(W, w, v):
    """Fonction permettant d'appliquer PLS a l'aide d'un ND-Tree
    Paramètres : W est la capacité du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères
    Retourne une approximation de l'ensemble des solutions efficaces ainsi que les points images de ces solutions
      dans l'espace des critères"""
```

1.1.2 Phase 2 - Élicitation incrémentale

Pour la phase 2, nous avons implémenté l'élicitation incrémentale avec trois fonctions d'agrégation possibles : la somme pondérée, OWA avec poids décroissants et l'intégrale de Choquet avec capacité super-modulaire.

```
PMR(x, y, v, preferences, fctAgregation):
    """Fonction permettant de calculer le PMR (Pairwise Max Regret) entre x et y pour un problème de maximisation
      avec la fonction d'agrégation
    fctAgregation
    Paramètres : x une solution
                 y une solution
                 v est la matrice de valuation des objets sur les différents critères
                 préférences une liste de couples (a,b) tels que a est préféré à b par le décideur (a et b sont
     des solutions)
                 fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
     intégrale de Choquet: "Ch")
    Retourne le PMR entre x et y"""
```

```
MMR(X, v, preferences, fctAgregation):
    """Fonction permettant de calculer le MMR (MiniMax Regret)
    Paramètres : X un ensemble de solutions
                 v est la matrice de valuation des objets sur les différents critères
                 preferences une liste de couples (a,b) tels que a est préféré à b par le décideur (a et b sont
     des solutions)
                 fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
     intégrale de Choquet: "Ch")
    Retourne le MMR dans X ainsi que la solution x pour laquelle ce minimum est atteint et la solution y qui fait
      atteindre le max dans le MR"""
```

```
generePoids(v, fctAgregation, verbose=False):
  """Fonction permettant de générer aléatoirement les poids des critères optimaux du décideur
  Paramètres : v est la matrice de valuation des objets sur les différents critères
         fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou intégrale
      de Choquet: "Ch")
               verbose indique si on veut afficher le déroulement (True) ou non (False)
  Retourne un vecteur de poids"""
```

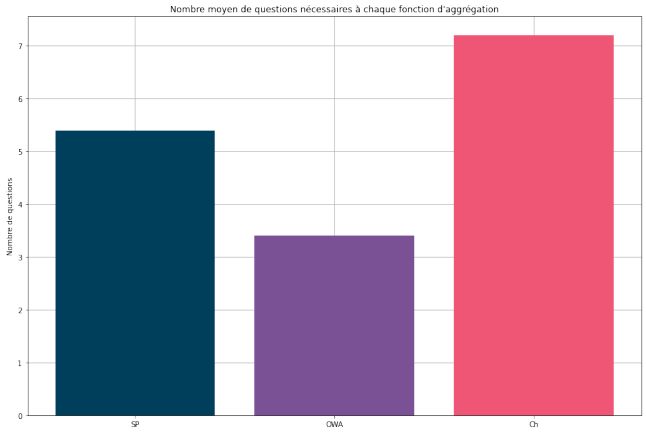
```
genereCapacite(nbCriteres, verbose=False):
  """Fonction permettant de générer aléatoirement les capacités convexes des critères optimaux du décideur
  Paramètres : nbCriteres est le nombre de critères à considérer
               verbose indique si on veut afficher le déroulement (True) ou non (False)
  Retourne un dictionnaire de capacités"""
```

```
CSS(X, v, preferences, fctAgregation):
    """Fonction permettant d'effectuer la Current Solution Strategy (CSS)
  Paramètres : X un ensemble de solutions
               v est la matrice de valuation des objets sur les différents critères
               preferences une liste de couples (a,b) tels que a est préféré à b par le décideur (a et b sont
    des solutions)
               fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
    intégrale de Choquet: "Ch")
    Retourne l'arrondi du MMR dans X ainsi que la solution x pour laquelle ce minimum est atteint et la solution
    y qui fait atteindre le max dans le MR"""
```

```
ElicitationIncrementale(X, v, fctAgregation, poids=None, preferences = None, verbose=False):
  """Fonction permettant d'effectuer l'élicitation incrémentale des poids des critères (en interaction avec l'
    utilisateur si poids null)
  Paramètres : X un ensemble de solutions
       v est la matrice de valuation des objets sur les différents critères
       fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou intégrale
    de Choquet: "Ch")
       poids permet de préciser le jeu de poids choisi par l'utilisateur si on veut faire la simulation (sans
    interaction)
       preferences permet de préciser des préférences déjà établies (utile pour PLS_Combine)
               verbose indique si on veut afficher le déroulement (True) ou non (False)
  Retourne une approximation de la solution optimale pour le décideur ainsi que le nombre de questions posées
    pour l'atteindre"""
```

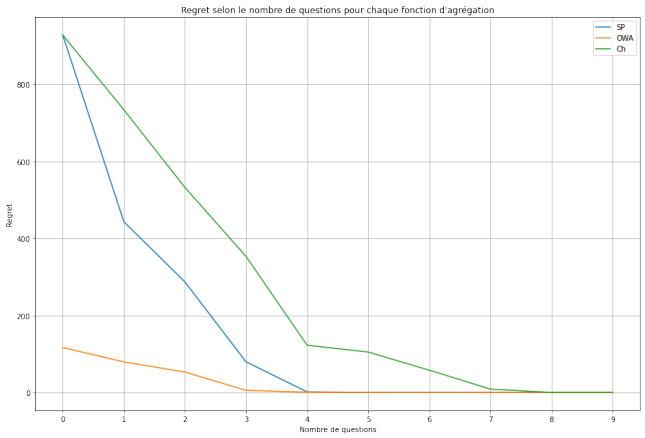
1.2 Question a

Les résultats suivants ont été menés sur un sous-ensemble de données dans lequel il y a 20 objets et 3 critères. On calcule la moyenne sur 20 essais. "SP" correspond à la somme pondérée et "Ch" à l'intégrale de Choquet.



On remarque tout d'abord que pour les trois fonctions d'agrégation présentées ci-dessus, on peut déterminer la solution optimale pour le décideur en un nombre acceptable de questions (en dessous de 8).
Ensuite, la différence du nombre de questions nécessaires en fonction de la fonction d'agrégation peut s'expliquer par des contraintes plus ou moins discriminantes déterminées par les fonctions d'agrégation elles-mêmes.

1.3 Question b



Comme attendu, plus on pose de questions, plus le regret minimax diminue puisqu'à chaque question posée, on élimine les solutions ne respectant pas la réponse (préférence) du décideur. Le regret encouru diminue donc au fil des questions. De plus, le regret minimax semble être plus important lorsque la fonction d'agrégation du décideur est l'intégrale de Choquet que lorsque celle-ci est la somme pondérée que lorsque celle-ci est l'OWA. Cela peut s'expliquer par des contraintes plus ou moins discriminantes déterminées par les fonctions d'agrégation elles-mêmes (de la même manière qu'à la question a, OWA est plus discriminante que la somme pondérée qui est plus discriminante que l'intégrale de Choquet).

2 Deuxième procédure de résolution

Nous avons implémenté la deuxième procédure d'élicitation avec trois fonctions d'agrégation possibles : la somme pondérée, OWA avec poids décroissants et l'intégrale de Choquet avec capacité super-modulaire. Nous avons également implémenté cette méthode avec des listes classiques et avec des ND-Tree pour stocker le front approximé de Pareto.

```
solutionInitiale(W, w, v):
    """Fonction initialisant la population pour le PLS_Combine
    La population initiale sera constituée d'une seule solution réalisable, générée comme dans l'article 2 (
     section 4.1), câd :
    tant qu'il reste de la place dans le sac, l'objet maximisant (somme de ses critères))/nbCriteres (de poids
     inférieur à la quantité de place
    restante) est ajouté
    Paramètres : W est la capacité du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères
    Retourne une solution initiale"""
```

```
PLS_Combine(W, w, v, fctAgregation, poids=None, verbose=False):
    """Fonction PLS combinant procédure d'élicitation incrémentale et recherche locale
    Paramètres : W est la capacité du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères
                 fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
     intégrale de Choquet: "Ch")
        poids permet de préciser le jeu de poids choisi par l'utilisateur si on veut faire la simulation (sans
     interaction)
                 verbose indique si on veut afficher le déroulement (True) ou non (False)
    Retourne la solution (optimale localement) à conseiller au décideur ainsi que le nombre de questions posées
     """
```

```
PLS_Combine_ND_Tree(W, w, v, fctAgregation, poids=None, verbose=False):
    """Fonction PLS combinant procédure d'élicitation incrémentale et recherche locale avec des ND-Tree
    Paramètres : W est la capacité du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères
                 fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
     intégrale de Choquet: "Ch")
        poids permet de préciser le jeu de poids choisi par l'utilisateur si on veut faire la simulation (sans
     interaction)
                 verbose indique si on veut afficher le déroulement (True) ou non (False)
    Retourne la solution (optimale localement) à conseiller au décideur ainsi que le nombre de questions posées
     """
```

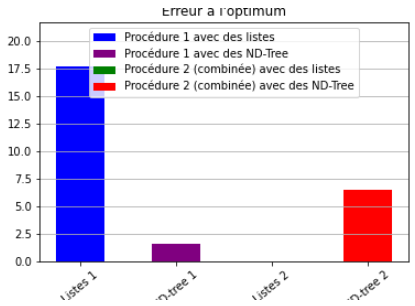
3 Comparaison des deux méthodes

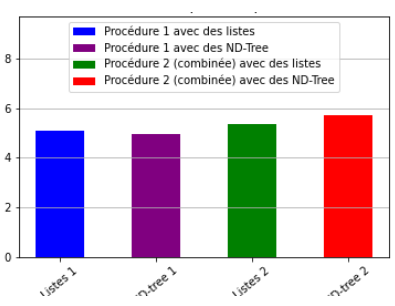
```
solutionOptimale(W, w, v, poids, fctAgregation):
    """Fonction permettant de déterminer la solution optimale selon la fonction d'agrégation et les poids en entr
     ée
    Paramètres : W est la capacité du sac
                 w est la liste de poids des objets
                 v est la matrice de valuation des objets sur les différents critères
                 poids le vecteur de poids du décideur
                 fctAgregation le nom de la fonction d'agrégation choisie (OWA: "OWA", somme pondérée: "SP", ou
     intégrale de Choquet: "Ch")
    Retourne la solution optimale parmi toutes les solutions possibles de l'instance"""
```

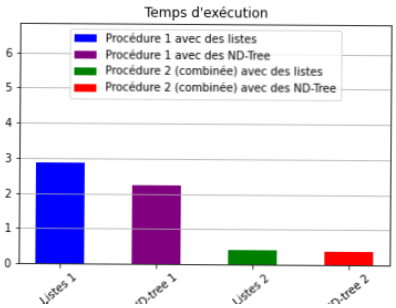
Les résultats suivants ont été menés sur un sous-ensemble de données dans lequel il y a 20 objets et 3 critères.

3.1 Somme pondérée

En faisant la moyenne des résultats obtenus pour 30 jeux de paramètres différents de la fonction d'agrégation, nous obtenons les résultats suivants.



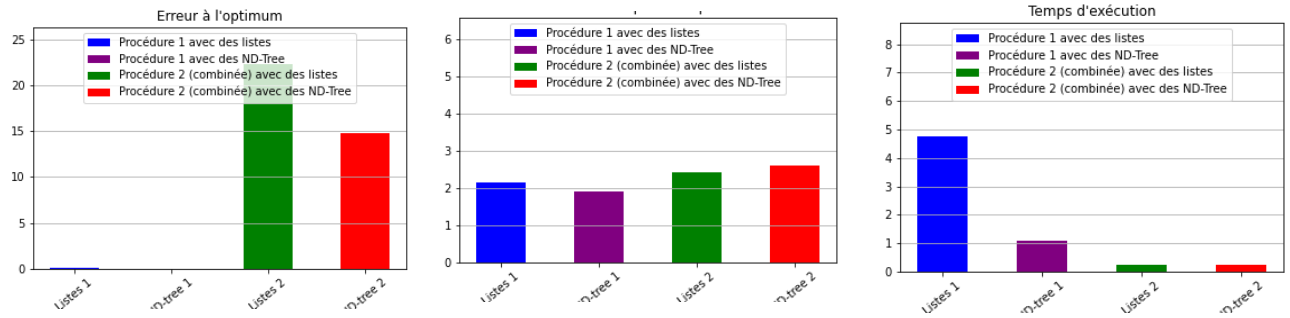




La première procédure avec des listes met en moyenne un temps de 2.8543 s , pose 5.1 question(s), et a une erreur de 17.7365
La première procédure avec des ND-Tree met en moyenne un temps de 2.2298 s , pose 4.95 question(s), et a une erreur de 1.6375
La deuxième procédure avec des listes met en moyenne un temps de 0.4234 s , pose 5.35 question(s), et a une erreur de 0.0
La deuxième procédure avec des ND-Tree met en moyenne un temps de 0.3879 s , pose 5.7 question(s), et a une erreur de 6.4535

3.2 OWA avec poids décroissants

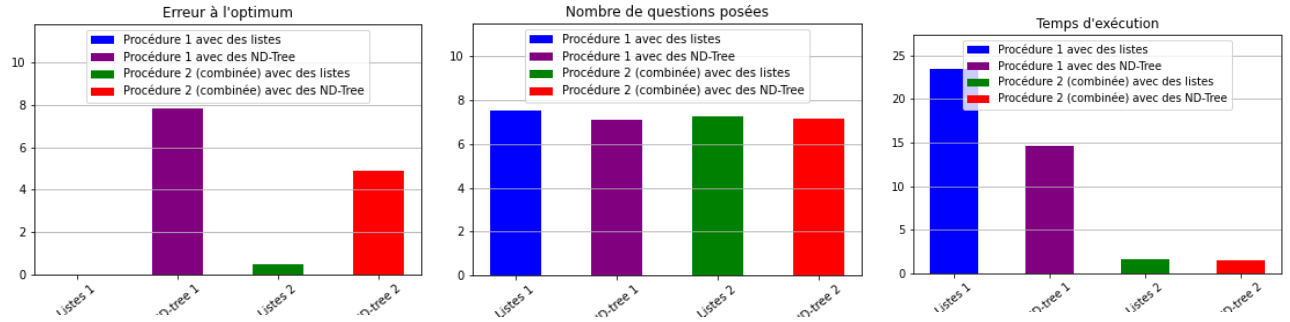
En faisant la moyenne des résultats obtenus pour 30 jeux de paramètres différents de la fonction d'agrégation, nous obtenons les résultats suivants.



La première procédure avec des listes met en moyenne un temps de 4.7691 s, pose 2.1666 question(s), et a une erreur de 0.083
La première procédure avec des ND-Tree met en moyenne un temps de 1.1015 s, pose 1.9 question(s), et a une erreur de 0.0
La deuxième procédure avec des listes met en moyenne un temps de 0.2270 s, pose 2.4333 question(s), et a une erreur de 22.3353
La deuxième procédure avec des ND-Tree met en moyenne un temps de 0.2417 s, pose 2.6 question(s), et a une erreur de 14.75866

3.3 Intégrale de Choquet avec capacité super-modulaire

En faisant la moyenne des résultats obtenus pour 30 jeux de paramètres différents de la fonction d'agrégation, nous obtenons les résultats suivants.



La première procédure avec des listes met en moyenne un temps de 23.4041 s, pose 7.5333 question(s), et a une erreur de 0.0
La première procédure avec des ND-Tree met en moyenne un temps de 14.6115 s, pose 7.1 question(s), et a une erreur de 7.8296
La deuxième procédure avec des listes met en moyenne un temps de 1.5851 s, pose 7.2666 question(s), et a une erreur de 0.48166
La deuxième procédure avec des ND-Tree met en moyenne un temps de 1.5255 s, pose 7.1666 question(s), et a une erreur de 4.8833

3.4 Analyse

Tout d'abord, au niveau de l'erreur à l'optimum, celle-ci reste très faible puisque les solutions ont des valeurs de l'ordre du millier. Une erreur de maximum 20 sur plusieurs milliers est donc très négligeable. En ce qui concerne la comparaison des différentes méthodes, nos expérimentations n'ont pas permis de définir une méthode clairement plus efficace que les autres en terme de distance à l'optimum.

Comme attendu, en général, le nombre de questions posées avec la première méthode est moins important que le nombre de questions posées avec la seconde méthode puisqu'à chaque fois, on choisit la question qui discrimine le plus (c'est-à-dire la question qui va supprimer le plus de solutions de l'ensemble des solutions non dominées). Dans le cas de la première méthode, les questions posées prennent en compte l'ensemble du front de Pareto approximé (l'ensemble des solutions non-dominées encore en lice), les questions choisies discriminent donc beaucoup sur l'ensemble du front. Au contraire, lors de la seconde méthode, les questions posées discriminent beaucoup, mais seulement dans l'ensemble des solutions considérées (voisines d'une solution), il se peut donc que globalement (si l'on considère l'ensemble des solutions que l'on va évaluer comme intéressantes au long de toute la procédure) la question posée soit peu discriminante (elle n'est discriminante que sur un sous-ensemble).

De plus, le temps d'exécution est plus court pour la deuxième méthode que pour la première. En effet, ce résultat était attendu puisque la seconde méthode est une procédure où élicitation incrémentale et recherche locale sont combinées ce qui a pour conséquence de réduire le nombre de solutions générées et donc le temps de calcul. De plus, notons que la structure ND-Tree permet d'avoir un temps d'exécution plus court qu'avec des listes classiques. En effet, la structure de données du ND-Tree permet de comparer les solutions entre elles beaucoup plus rapidement (sans avoir à parcourir toute la liste des solutions et sans avoir à comparer toutes les solutions du front).

Notons également que cette structure de donnée est très efficace pour la première méthode et pas tellement (voire pas du tout) pour la deuxième méthode. En effet, ce comportement est dû au fait qu'avec la première méthode, on utilise un seul ND-Tree pour toute la procédure de recherche d'approximation du front de Pareto, le ND-Tree permet donc de faire les comparaisons plus vite avec un nouveau vecteur de valuation alors que dans la seconde méthode, on utilise un nouveau ND-Tree ne contenant que la solution de laquelle on veut générer le voisinage à chaque itération et à chaque itération, on veut qu'à la fin de l'itération, il ne reste qu'un seul élément (duquel on va générer un voisinage à l'itération d'après), le fait de supprimer autant d'éléments et d'en rajouter autant à chaque itération est très coûteux en temps. De plus, le gain de temps associé aux comparaisons n'est pas très représentatif puisqu'à chaque itération, on a un ensemble relativement petit d'éléments dans le ND-Tree.