



MASTER AGENTS DISTRIBUÉS, ROBOTIQUE, RECHERCHE OPÉRATIONNELLE,
INTERACTION, DÉCISION

MADMC - Projet MADMC 2023–2024

Rapport de projet

Tony HU - Osmane El Montaser
21212455 - 21201287

Sommaire

1	Introduction	1
1.1	Format des données	1
2	Première procédure de résolution	1
2.1	Recherche locale de Pareto	1
2.1.1	Initialisation de points	1
2.1.2	Fonction de voisinage	2
2.1.3	Structure de données pour la mise à jour : ND-Tree	3
2.1.4	Déroulement de l'algorithme	4
2.1.5	Visualisation de la recherche locale	5
2.2	Elicitation incrémentale	5
2.2.1	Decision Maker	5
2.2.2	Initialisation du modèle	7
2.2.3	Ajout d'une question	7
2.2.4	Calcul du PMR et optimisation	10
2.2.5	Calcul du regret minimax (MMR)	11
2.2.6	Current Solution Strategy	11
2.2.7	Procédure	11
2.3	Etudes des résultats	12
3	Deuxième procédure de résolution	15
3.1	Implémentation	15
3.2	Données	15
4	Données et comparaison des deux méthodes	16
4.1	Obtention de la solution optimale	16

4.1.1	Somme pondérée	16
4.1.2	OWA	17
4.1.3	Choquet	17
4.2	Comparaison des procédures	17
4.2.1	Temps de calcul	18
4.2.2	Erreur par rapport à la solution optimale du décideur	19
4.2.3	Nombre de questions posées	20

1 Introduction

On s'intéresse dans ce projet à l'approximation des solutions optimales non-dominées d'un problème de sac-à-dos multi-objectifs (dans le cas d'une maximisation).

1.1 Format des données

```
w=np.zeros(200, dtype=int)
v=np.zeros((200,6), dtype=int)
filename = "data/2KP200-TA-0.dat"
W = readFile(filename,w,v)

#Reshape v and w to only take first the required number of objects and
#criteria

v = v[:N_OBJETS]
v = v[:, :N_CRITERES]
w = w[:N_OBJETS]

#W equals the sum of the weights divided by 2
W = int(w.sum()/2)
```

Voici un extrait du code permettant de lire nos données pour le problème du sac-à-dos. **w** correspond au poids de chaque objet, **v**, la valeur sur chaque critère de chaque objet (ici, nous avons 6 critères disponibles.) et **W**, correspondant à la capacité totale que peut supporter le sac-à-dos. La capacité est donnée par la formule suivante : $\left\lfloor \frac{\sum_{i=1}^n w_i}{2} \right\rfloor$

2 Première procédure de résolution

Dans cette première partie, nous vous présentons la mise en œuvre d'une procédure en deux phases pour déterminer la solution préférée du décideur. Cette procédure se déroule de la façon suivante : nous déterminons une approximation des points non-dominés (au sens de Pareto), puis nous générons la solution préférée du décideur parmi les points non-dominés à l'aide d'une procédure d'élitisation incrémentale.

2.1 Recherche locale de Pareto

2.1.1 Initialisation de points

Dans un premier temps, il est nécessaire d'initialiser des points qui vont ensuite être utilisés pour démarrer la procédure de recherche locale. Nous utilisons l'algorithme suivant pour récupérer les solutions initiales :

```

#W equals the sum of the weights divided by 2
W = int(w.sum()/2)

costs = {}
solutions = {}

for i in range(m):
    w_total = 0
    current_solution = set()
    current_cost = np.zeros(v.shape[1])
    arr = np.arange(w.shape[0])
    np.random.shuffle(arr)
    for j in arr:
        if w_total + w[j] <= W:
            current_solution.add(j)
            next_item = v[j]
            w_total += w[j]
            for k in range(v.shape[1]):
                current_cost[k] += next_item[k]
    costs[i] = current_cost

    solutions[i] = current_solution

```

Les variables **costs** et **solutions** sont des dictionnaires qui pour chaque clé (une solution trouvée), permet de retourner le coût/valeur de la solution pour le dictionnaire **costs**, et les objets de la solution pour le dictionnaire **solutions**.

À noter que ces solutions ne sont pas tous pareto non dominées, nous récupérons les solutions non dominées après, lors de l'insertion de nos points dans notre structure de donnée ND-Tree, où nous ferons la vérification du front de Pareto.

2.1.2 Fonction de voisinage

```

#Donne les voisins de la solution sol, dont la valeur de la solution est cost,
#pour la liste d'objet obj et leur poids weight
def voisinage(cost, sol, obj, weight, max_weight,):
    inside = np.array(list(sol))
    outside = np.delete(np.array(list(range(len(obj)))), inside)

    w_sol = weight[np.array(list(sol))].sum()
    voisinage = dict()
    voisinage_cost = dict()
    k = 0
    for i in inside:
        for j in outside:
            if w_sol - weight[i] + weight[j] <= max_weight:
                new_sol = sol.copy()
                new_sol.remove(i)
                new_sol.add(j)

```

```
voisinage[k] = set(new_sol)
voisinage_cost[k] = cost - obj[i] + obj[j]
k+=1
return voisinage_cost, voisinage
```

Notre fonction de voisinage retourne tous les voisins de la solution donnée en entrée. La fonction de voisinage consiste à retirer un objet de la solution courante **sol** et à ajouter un nouvel objet (échange 1-1). Nous testons toutes les combinaisons qui respectent la contrainte de capacité du sac-à-dos.

Nous avons également une variante de cette fonction de voisinage, qui permet de retourner que des solutions qui n'ont pas déjà été visité. Cette variante est utilisée pour l'algorithme de recherche locale.

2.1.3 Structure de données pour la mise à jour : ND-Tree

Nous avons utilisé la structure ND-Tree[1] afin de mettre à jour notre front de pareto. Nous avons recodé la structure et les différentes fonctions dans le fichier **ndtree.py** de notre code source.

Étant dans le cadre d'une maximisation, nous avons utilisé la fonction suivante pour comparer deux vecteurs de coûts **u** et **v**

```
#Retourne true si u domine v, false sinon
def pareto_dominance_maximisation(u, v):
    return np.all(u >= v) and np.any(u > v)
```

2.1.4 Déroulement de l’algorithme

```

#Pareto local search algorithm
#Entree: costs, solutions, v, w, W.
#costs et solutions sont la population initiale choisie pour l'algorithme,
#v et w sont les donnees pour les objets (valeurs et poids)
#W le poids maximal du sac
def PLS(costs, solutions, v, w, W, do_display = False):
    visited = set()
    tree = NDTree()
    for k, val in costs.items():
        tree.update_tree(val, solutions[k]) #Initialisation of the tree with
                                           #pareto dominating solutions
    costs, solutions = tree.get_all_costs_values()
    for k, val in costs.items():
        visited.add(frozenset(val))
    empty = False

    while not empty:
        print("Current number of solutions: " + str(len(costs)))
        id, cost = costs.popitem()
        sol = solutions.pop(id)
        to_test, to_test_sol, visited = voisinage_var(cost, sol, v, w, W,
                                                       visited)

        if len(to_test) == 0:
            empty = True
        else:
            for k, val in to_test.items():
                tree.update_tree(val, to_test_sol[k])
            costs, solutions = tree.get_all_costs_values()

        #Display the pareto front for the first two objectives dynamically (
        #for notebook)

        if do_display == True:
            ...

        if len(costs) == 0:
            empty = True

    costs, solutions = tree.get_all_costs_values()
    return costs, solutions

```

La première étape de l’algorithme de recherche local est d’initialiser notre **NDTree** avec la population initiale que nous avons généré précédemment. Nous initialisons un set **visited** qui correspond aux solutions déjà visitées et nous mettons à jour notre **NDTree** vierge avec notre population de solutions. Après l’ajout de chaque solution, il en résulte un **NDTree** dont on peut récupérer nos nouveaux **costs** et **solutions** qui correspondent aux solutions non-dominées parmi les solutions insérées précédemment.

Nous procédons ensuite itérativement à chercher les voisins d’une solution de notre ensembles **costs** et mettre à jour notre **NDTree** selon les voisins trouvés jusqu’à que nous ayons plus de nouveaux noeuds à visiter.

Nous récupérons donc, à la fin de l'algorithme, une approximation de l'ensemble de solutions non-dominées.

2.1.5 Visualisation de la recherche locale

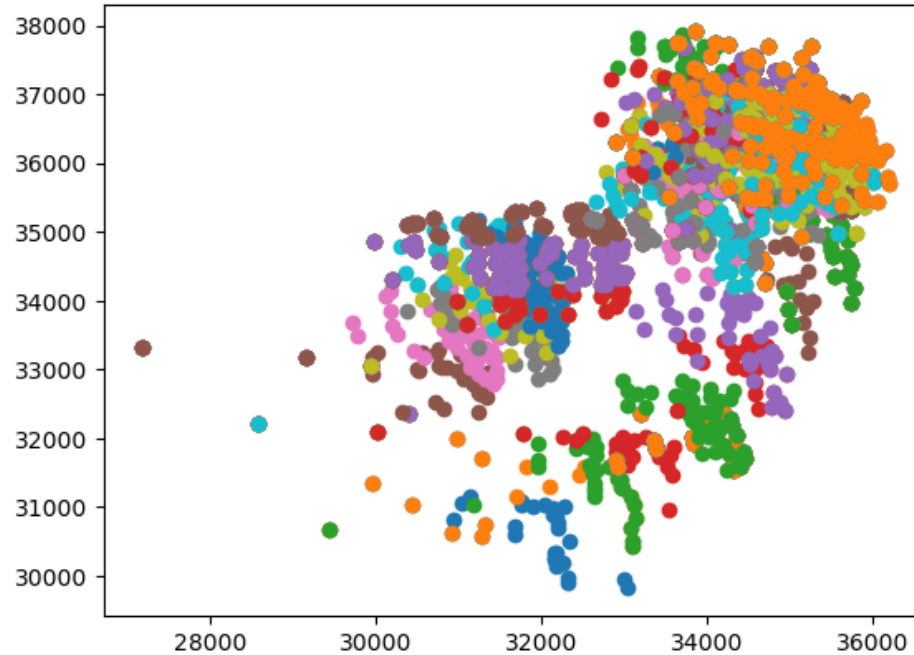


FIGURE 1 – Visualisation de l'algorithme de recherche local pour 100 objets et 3 critères (sur les 2 premiers critères)

Voici la représentation graphique des différents points explorés lors de la recherche locale. Nous représentons ici seulement les deux premiers critères pour une recherche sur trois critères, ce qui explique pourquoi nous n'avons pas un front de pareto visible. Étant dans une maximisation, on observe la valeur des solutions explorées augmente correctement (les premiers points étant ceux en bas à gauche).

2.2 Elicitation incrémentale

2.2.1 Decision Maker

Afin de pouvoir poser les questions pour identifier la solution optimale d'un décideur, nous avons créé une classe **DM**


```

#Decision Maker, used to ask queries
class DM():

    def __init__(self, nb_criteres, agr_func):
        self.n_crit = nb_criteres
        self.agr_func = agr_func
        self.comb = get_all_combinations(self.n_crit)
        self.weights = self.get_random_weights()

    #Return a random vector of weights of size n_crit, summing to 1
    def get_random_weights(self):
        if self.agr_func == "WS":
            return np.random.dirichlet(np.ones(self.n_crit),size=1)[0]
        elif self.agr_func == "OWA": #OWA with decreasing weights
            weights = np.random.dirichlet(np.ones(self.n_crit),size=1)[0]
            weights = np.sort(weights)
            return weights[::-1]
        elif self.agr_func == "Choquet":
            #Generate a random convex Choquet integral
            weights_len = np.zeros(self.n_crit)
            weights = []
            for i in range(1, self.n_crit):
                list_i = [l for l in self.comb if len(l) == i]
                total = 0
                for _ in list_i:
                    r = random.random() + weights_len[i-1]
                    weights.append(r)
                    total += r
                weights_len[i-1] += total
                weights_len[i] += weights_len[i-1]

            last = [l for l in self.comb if len(l) == self.n_crit] #last
                                                                #element, which is v(N)
            for _ in last:
                r = random.random() + weights_len[self.n_crit-1]
                weights.append(r)
            #Normalize weights
            weights = np.array(weights)
            return weights / weights.sum()

```

- Nous partons d'un vecteur de poids au hasard pour chacune de nos fonctions d'agréations.
- Pour la somme pondérée : nous créons un vecteur poids normalisé.
 - Pour OWA : nous créons un vecteur poids normalisé qui est triée de façon décroissante.
 - Pour l'intégrale de Choquet : nous créons un vecteur poids qui correspond à une capacité super-modulaire/convexe. Pour s'assurer de sa convexité, nous construisons notre capacité telle que pour chaque "niveau" de combinaison de critères, les poids des combinaisons de ce niveau soit supérieur à la somme des poids des niveaux précédents. On retourne ensuite la capacité normalisée.

Nous avons également une autre fonction appartenant à la classe, qui permet de comparer deux solutions x et y , s'il préfère x à y . Cette fonction **compare** est ensuite utilisé lors de notre élicitation incrémentale.

```
def compare(self, x, y):
    if self.agr_func == "WS":
        return np.sum(self.weights*x) >= np.sum(self.weights*y)
    elif self.agr_func == "OWA":
        x_t = np.sort(x)
        y_t = np.sort(y)
        return np.sum(self.weights*x_t) >= np.sum(self.weights*y_t)
    elif self.agr_func == "Choquet":
        return self.get_choquet_value(x) >= self.get_choquet_value(y)
```

2.2.2 Initialisation du modèle

Afin de calculer les regrets maximum utilisés pour notre élicitation incrémentale, nous utilisons le solveur gurobi pour pouvoir les calculer. On crée le modèle utilisé par Gurobi dans une classe nommée : **RBE**.

On initialise les variables de notre modèle à l'aide de notre fonction :

```
def init_model(self):
    if self.agr_func == "WS" or self.agr_func == "OWA":
        self.weight_var = [self.model.addVar(vtype=gp.GRB.CONTINUOUS, lb=0, ub
                                             =1, name="w"+str(w)) for w in
                           range(self.n_crit)]
        self.model.addConstr(gp.quicksum(self.weight_var) == 1)
    elif self.agr_func == "Choquet":
        #We add as many variables as there are subsets of criteria
        self.weight_var = [self.model.addVar(vtype=gp.GRB.CONTINUOUS, lb=0, ub
                                             =1, name="w"+str(w)) for w in
                           range(2**self.n_crit-1)]
```

Nous avons au total **n_crit** variable pour le poids dans le cas de la somme pondérée et du OWA, et **2**n_crit-1** variables dans le cas de l'intégrale de Choquet. Ce nombre représente toutes les combinaisons de critères possibles dans notre capacité pour Choquet.

2.2.3 Ajout d'une question

Les questions posés pour la prise de décision sont des comparaisons de deux solutions. On demande au décideur s'il préfère une solution x à une solution y.

Pour ajouter notre paire de solutions (x, y) et les contraintes qu'elles impliquent dans notre modèle, nous utilisons la fonction :

```

def insert_pair(self, pair):
    self.pairs.append(pair)
    self.n_queries += 1
    x = pair[0]
    y = pair[1]
    if self.agr_func == "WS":
        self.model.addConstr(gp.quicksum(self.weight_var[i]*(x[i]-y[i]) for i
                                         in range(self.n_crit)) >= 0)

    elif self.agr_func == "OWA":
        #sort x and y
        x_temp = np.sort(x)
        y_temp = np.sort(y)
        self.model.addConstr(gp.quicksum(self.weight_var[i]*(x_temp[i]-y_temp[
                                         i]) for i in range(self.n_crit)
                                         ) >= 0)

    elif self.agr_func == "Choquet":
        #sort x and y
        x_temp = np.sort(x)
        y_temp = np.sort(y)
        weights_x = self.get_weight_choquet(x)
        weights_y = self.get_weight_choquet(y)
        e = gp.LinExpr(self.weight_var[weights_x[0]]*x_temp[0] - self.
                        weight_var[weights_y[0]]*y_temp
                        [0])

        self.model.addConstr((e + gp.quicksum(self.weight_var[weights_x[i]]*(
                                         x_temp[i]-x_temp[i-1]) - self.
                                         weight_var[weights_y[i]]*(
                                         y_temp[i]-y_temp[i-1]) for i in
                                         range(1,self.n_crit))) >= 0 )

    self.add_convexity_constraint()

```

Ainsi, pour la somme pondérée et OWA, nous devons rajouter la contrainte $\sum w_i(x_i - y_i) \leq 0$, car nos fonctions d'agrégations f doivent être consistant avec chaque pair que nous rajoutons dans notre modèle, tel que $f(x) \geq f(y)$ pour toute f appartenant à la famille de fonction d'agrégations.

Nous avons un format différent pour Choquet. Nous ajoutons également les contraintes de convexités pour les poids.

```

#Add constraints to the model to ensure convexity for the Choquet integral
def add_convexity_constraint(self):
    added = set()
    for i in self.comb:
        for j in self.comb:
            #if i != j and the pair is not already in added
            if i != j and (frozenset(i),frozenset(j)) not in added and (
                frozenset(j),frozenset(i))
                not in added:
                added.add((frozenset(i),frozenset(j)))
                A_U_B = list(set(i).union(set(j)))
                A_I_B = list(set(i).intersection(set(j)))
                if len(A_I_B) == 0:
                    self.model.addConstr(self.weight_var[self.comb.index(A_U_B
                        )] >= self.
                        weight_var[self.
                        comb.index(i)] +
                        self.weight_var[
                        self.comb.index(j)]
                        )
                else:
                    self.model.addConstr(self.weight_var[self.comb.index(A_U_B
                        )] + self.
                        weight_var[self.
                        comb.index(A_I_B)]
                        >= self.weight_var[
                        self.comb.index(i)]
                        + self.weight_var[
                        self.comb.index(j)]
                        )

```

2.2.4 Calcul du PMR et optimisation

```
#The Pairwise Max Regret (PMR) of alternatives x, y in X, such as x is known
to be preferred to y
def PMR(self, x, y):
    if self.agr_func == "WS":
        self.model.setObjective(gp.quicksum(self.weight_var[i]*(y[i]-x[i]) for
                                             i in range(self.n_crit)), gp.
                                             GRB.MAXIMIZE)

    elif self.agr_func == "OWA":
        #sort x and y
        x_temp = np.sort(x)
        y_temp = np.sort(y)
        self.model.setObjective(gp.quicksum(self.weight_var[i]*(y_temp[i]-
                                             x_temp[i]) for i in range(self.
                                             n_crit)), gp.GRB.MAXIMIZE)

    elif self.agr_func == "Choquet":
        #sort x and y
        x_temp = np.sort(x)
        y_temp = np.sort(y)
        weights_x = self.get_weight_choquet(x)
        weights_y = self.get_weight_choquet(y)
        first = self.weight_var[weights_y[0]]*y_temp[0] - self.weight_var[
                                             weights_x[0]]*x_temp[0]

        #add first to the objective
        self.model.setObjective(first + gp.quicksum(self.weight_var[weights_y[
                                             i]]*(y_temp[i]-y_temp[i-1]) -
                                             self.weight_var[weights_x[i]]*(
                                             x_temp[i]-x_temp[i-1]) for i in
                                             range(1,self.n_crit)), gp.GRB.
                                             MAXIMIZE)

    self.model.update()
    self.model.optimize()
    return self.model.objVal
```

Afin de désormais calculer le PMR de chaque pair que nous allons tester afin de trouver le MMR, nous avons plus qu'à mettre notre objectif dans notre modèle. Nous mettons ainsi à jour notre modèle seulement lorsque l'on doit redéfinir l'objectif lors du test d'une paire pour le PMR, sauf dans le cas où on l'on insère une nouvelle question dans notre modèle (on doit mettre à jour nos contraintes).

Après avoir rajouté notre objectif, nous mettons à jour le modèle, et retournons la valeur de l'objectif après optimisation.

2.2.5 Calcul du regret minimax (MMR)

```
#The Minimax Regret (MMR) over X
def MMR(self, X):
    all_mr = [self.MR_normalized(x, X) for x in X]
    ind = np.argmin(all_mr)
    return all_mr[ind], X[ind]

#The Max Regret (MR) of alternative x in X
def MR(self, x, X):
    return np.max([self.PMR(x, y) for y in X])

#The Max Regret normalized, used for the incremental elicitation
def MR_normalized(self, x, X):
    return np.max([self.PMR(x, y) for y in X])/self.init_MR
```

2.2.6 Current Solution Strategy

```
#Current Solution strategy (CSS), return xp and yp for the next query
def CSS_get_compare(self, X):
    xp = np.argmin([self.MR(x, X) for x in X])
    yp = np.argmax([self.PMR(X[xp], y) for y in X])
    return X[xp], X[yp]

#Ask the DM to compare two new alternatives
def CSS_ask_query(self, X, DM):
    xp, yp = self.CSS_get_compare(X)
    if DM.compare(xp, yp):
        self.insert_pair([xp, yp])
    else:
        self.insert_pair([yp, xp])
    return self.MMR(X)
```

Afin de choisir la paire (x, y) , correspondant à la question que nous insérons dans le modèle, nous utilisons la méthode CSS proposée par l'article [Projet1.pdf\[2\]](#). Nous demandons au **DM** de comparer notre pair (x, y) choisi avec la méthode CSS, et insérons la nouvelle paire dans notre modèle. Cette insertion correspond ainsi à une question posée au décideur.

2.2.7 Procédure

La procédure d'élicitation incrémentale se déroule de la façon suivante. En partant d'une liste de solutions approchées non-dominées du front de pareto obtenue précédemment, nous posons itérativement des questions jusqu'à que le regret minimax retourné en le calculant avec Gurobi soit supérieur à un **EPS** fixé, qui correspond à notre minimum que l'on souhaite au moins obtenir. En fixant **EPS** proche de 0, nous continuons de poser des questions qui a pour conséquence de nous permettre de nous rapprocher de la meilleure solution pour le décideur parmi la liste de nos solutions approchées.

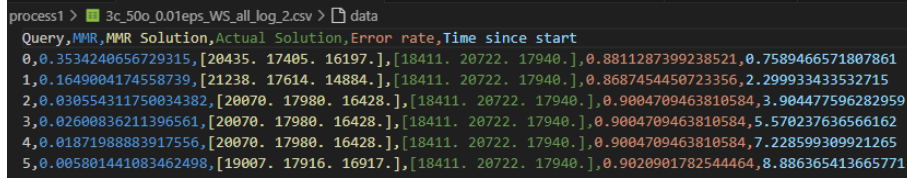
```

print("Starting Incremental Elicitation ... \n")
#Incremental Elicitation
while MMR_value > EPS:
    res = rbe.CSS_ask_query(X, dm)
    MMR_value = res[0]
    print("Query " + str(rbe.n_queries) + ": " + str(rbe.pairs[-1]))
    print("MMR value for the query: " + str(res[0]) + ", found for the
          solution " + str(res[1]) + "\n")

```

2.3 Etudes des résultats

Nous avons sauvegardé nos résultats dans des fichier **.csv**. Ce fichier stocke pour chaque question posée le MMR à cette question, la solution retournée par le modèle, la solution optimale (dont on décrit la méthode d'obtention dans la partie 4), l'error rate (ou le rapport valeur solution modèle / valeur solution optimale), et le temps d'exécution pour la partie élicitation incrémentale.



```

process1 > 3c_50o_0.01eps_WS_all_log_2.csv > data
Query,MMR,MMR Solution,Actual Solution,Error rate,Time since start
0,0.3534240656729315,[20435. 17405. 16197.],[18411. 20722. 17940.],0.8811287399238521,0.7589466571807861
1,0.1649004174558739,[21238. 17614. 14884.],[18411. 20722. 17940.],0.8687454450723356,2.299933433532715
2,0.030554311750034382,[20070. 17980. 16428.],[18411. 20722. 17940.],0.9004709463810584,3.904477596282959
3,0.02600836211396561,[20070. 17980. 16428.],[18411. 20722. 17940.],0.9004709463810584,5.570237636566162
4,0.01871988883917556,[20070. 17980. 16428.],[18411. 20722. 17940.],0.9004709463810584,7.228599309921265
5,0.005801441083462498,[19007. 17916. 16917.],[18411. 20722. 17940.],0.9020901782544464,8.886365413665771

```

FIGURE 2 – Contenu d'un fichier log pour la première procédure

À partir des données obtenues sur 20 exécutions de la procédure pour 3 critères et 50 objets, nous avons les résultats suivants :

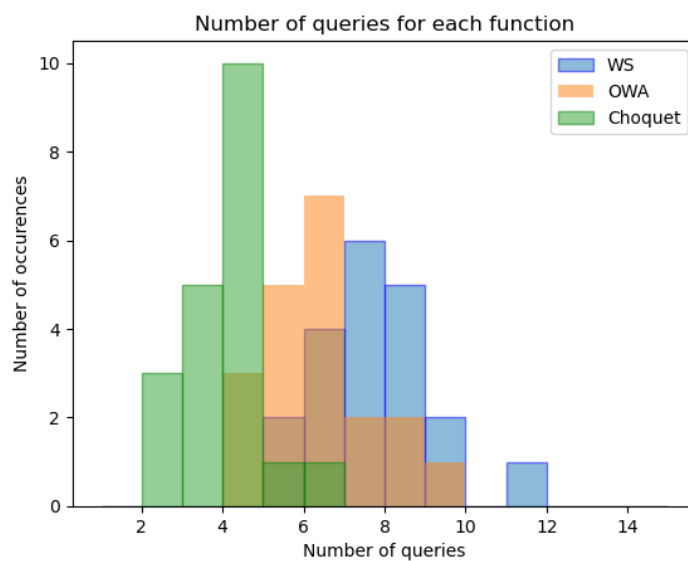


FIGURE 3 – Répartition du nombre de questions nécessaire pour chaque fonction

```

Nombres de questions moyens pour WS : 7.25, médiane : 7.0, écart-type : 1.4097872179871684
Nombres de questions moyens pour OWA : 5.9, médiane : 6.0, écart-type : 1.3379088160259653
Nombres de questions moyens pour Choquet : 3.6, médiane : 4.0, écart-type : 0.9695359714832658

```

FIGURE 4 – Comparaison des différentes fonctions

On remarque que le nombre de questions nécessaire est le plus faible avec des fonctions d'agréations de type Choquet, suivi de OWA, et enfin de somme pondéré.

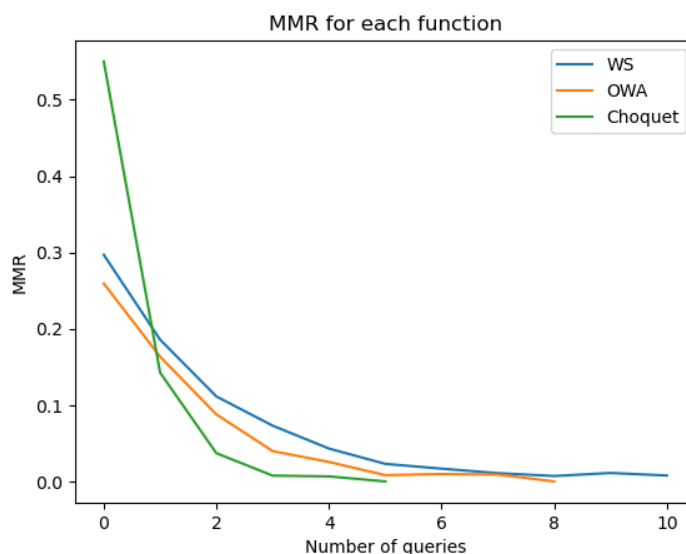


FIGURE 5 – Etude de la variation du regret minimax en fonction du nombre de questions posées pour les fonction d'agrégation

En étudiant également la variation du regret minimax en fonction du nombre de questions posées, il est évident que poser des questions pour les fonctions d'agrégations de type Choquet donne plus d'informations sur la solution optimale pour le décideur, d'où le fait que moins de questions sont nécessaire avant d'arriver à la solution optimale pour le décideur.

3 Deuxième procédure de résolution

La deuxième procédure de résolution est une procédure où élicitation incrémentale et recherche locale sont combinées afin de réduire le nombre de solutions générées.

3.1 Implémentation

Pour notre implémentation de cette deuxième procédure, nous sommes partis de la même liste de solution non-dominée générée aléatoirement. Cependant, au lieu de choisir une solution à visiter au hasard, nous choisissons la meilleure solution de la liste avec notre procédure d'élicitation incrémentale.

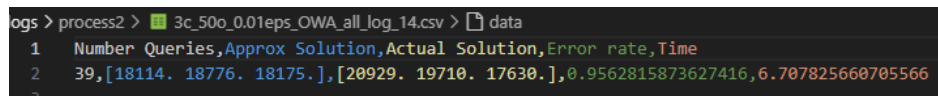
```
...
    to_visit, n_query = get_best_sol(X, dm, v.shape[1], agr_func, eps, logging
                                   ) #to_visit, la solution choisi
...

def get_best_sol(X, dm, n_crit, agr_func, eps, logging = False):
    rbe = RBE(n_crit, agr_func, X)
    res = rbe.MMR(X)
    MMR_value = res[0]
    i = 0
    while MMR_value > eps:
        i += 1
        if logging:
            sys.stdout.write("\rQuery " + str(i))
            sys.stdout.flush()
        res = rbe.CSS_ask_query(X, dm)
        MMR_value = res[0]
    return res[1], i
```

Nous continuons de visiter et de choisir une nouvelle solution jusqu'à que l'on trouve une meilleure solution avec les mêmes valeurs que l'ancienne[3].

3.2 Données

Le fichier stocké pour la deuxième procédure diffère de la première : nous avons plus qu'une ligne, qui correspond aux nombres de questions posées au total lors de la procédure, la solution retournée par le modèle, la solution optimale, l'error rate, et le temps d'exécution pour toute la procédure. Nous étudions les données obtenues à la prochaine partie.



```
ogs > process2 > 3c_50a_0.01eps_OWA_all_log_14.csv > data
1  Number Queries,Approx Solution,Actual Solution,Error rate,Time
2  39,[18114. 18776. 18175.],[20929. 19710. 17630.],0.9562815873627416,6.707825660705566
3
```

FIGURE 6 – Contenu d'un fichier log pour la deuxième procédure

4 Données et comparaison des deux méthodes

4.1 Obtention de la solution optimale

Afin de comparer les solutions approchées que nous avons obtenues à la vraie solution optimale du décideur, nous devons obtenir cette solution en optimisant un programme linéaire avec comme objectif la fonction d'agrégation du décideur.

4.1.1 Somme pondérée

```
#Get optimal solution for a weighted sum function using Gurobi
def get_opt_ws(v, w, W, weights): #v value of each object, w weight of each
                                #object, W max weight of the knapsack,
                                #weights weights of the Choquet integral

    model = gp.Model("ws")
    model.Params.LogToConsole = 0
    x = [model.addVar(vtype=gp.GRB.BINARY, name="x"+str(i)) for i in range(v.
                                         shape[0])]
    model.setObjective(gp.quicksum(weights[j]*v[i][j]*x[i] for i in range(len(x))
                                   ) for j in range(len(weights))), gp.GRB
                      .MAXIMIZE)
    model.addConstr(gp.quicksum(w[i]*x[i] for i in range(v.shape[0])) <= W)
    model.update()
    model.optimize()
    objets = []
    for v in model.getVars(): #Only var is x
        if v.X == 1:
            objets.append(v.varName[1:])
    model.write('modelws.LP')
    return (objets, model.objVal)
```

Le programme linéaire pour la somme pondérée est simple : nous ajoutons directement pour l'objectif la valeur obtenue selon les poids de la fonction d'agrégation pour chaque objet. On rajoute ensuite la contrainte qui est que les poids des objets choisis ne doivent pas dépasser la capacité maximale du sac à dos, puis on peut optimiser.

4.1.2 OWA

```
#Get optimal solution for a ordered weighted average function using Ogryczak
formulation, weights sorted in
descending order (monotonic)

def get_opt_owa(v, w, W, weights):
    model = gp.Model("owa")
    model.Params.LogToConsole = 0
    w_p = [weights[i] - weights[i+1] for i in range(len(weights)-1)] + [weights[-1]]

    m = len(w_p)
    r = [model.addVar(vtype=gp.GRB.CONTINUOUS, name="r"+str(k)) for k in range(m)]

    d = [[model.addVar(vtype=gp.GRB.CONTINUOUS, lb=0, name="d"+str(i)+"_"+str(k))
           for i in range(m)] for k in range(m)]
    x = [model.addVar(vtype=gp.GRB.BINARY, name="x"+str(i)) for i in range(v.shape[0])]
    model.setObjective(gp.quicksum((k+1)*w_p[k]*r[k] - gp.quicksum(w_p[k]*d[i][k]
        for i in range(m)) for k in range(m))
        , gp.GRB.MAXIMIZE)

    model.addConstr(gp.quicksum(w[i]*x[i] for i in range(v.shape[0])) <= W)
    for i in range(m):
        yi = gp.quicksum(v[:,i] * x)
        for k in range(m):
            model.addConstr(d[i][k] >= r[k] - yi)
    model.update()
    model.optimize()
    objets = []
    for v in model.getVars():
        if v.X == 1 and v.varName[0] == 'x':
            objets.append(v.varName[1:])
    model.write('modelowa.LP')
    return (objets, model.objVal)
```

Pour le OWA, nous utilisons la formule de Ogryczak[4] afin de faire notre programme linéaire.

4.1.3 Choquet

Nous n'avons pas pu réaliser le programme linéaire pour l'intégrale de Choquet, nous n'aurons donc pas les résultats pour le taux d'erreur sur l'intégrale de Choquet.

4.2 Comparaison des procédures

Pour comparer nos procédures, nous avons réalisé 20 itérations sur 3 critères et 50 objets de chaque procédure sur chaque type de fonction d'agrégation.

4.2.1 Temps de calcul

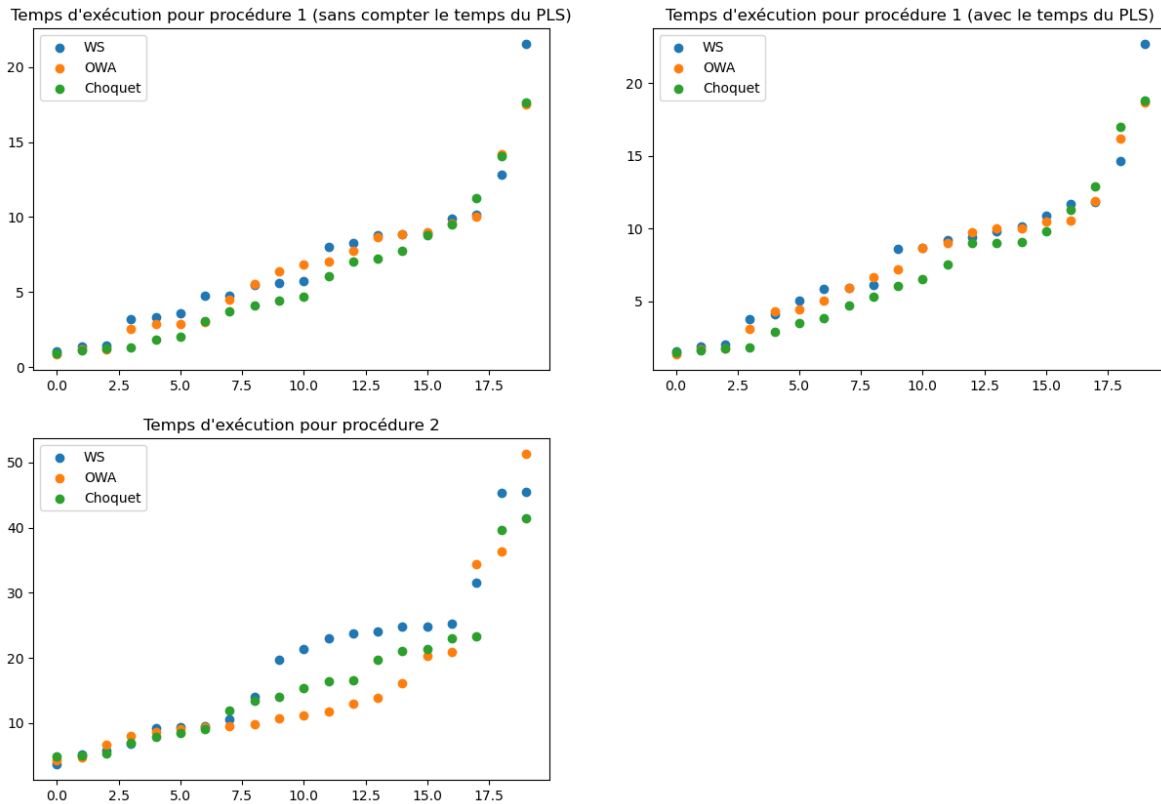


FIGURE 7 – Temps d'exécutions pour 20 procédures, triée du plus court au plus long

```

Temps d'exécution (procédure 1 avec PLS) moyen pour WS : 8.202742767333984, médiane : 8.63133716583252, écart-type : 4.882823236646262
Temps d'exécution (procédure 1 avec PLS) moyen pour OWA : 7.839288389682769, médiane : 7.944116115570068, écart-type : 4.51570892275163
Temps d'exécution (procédure 1 avec PLS) moyen pour Choquet : 7.212366914749145, médiane : 6.315677881240845, écart-type : 4.860250162444649
Temps d'exécution (procédure 2) moyen pour WS : 19.183842754364015, médiane : 20.556724309921265, écart-type : 11.934950344348808
Temps d'exécution (procédure 2) moyen pour OWA : 15.523044323921203, médiane : 10.963591694831848, écart-type : 11.730194548008987
Temps d'exécution (procédure 2) moyen pour Choquet : 16.262800550460817, médiane : 14.70840609073639, écart-type : 10.00578976262818

```

FIGURE 8 – Comparaison du temps d'exécution obtenue pour chaque procédure

Nous pouvons remarquer que le temps d'exécution de la deuxième procédure est plus bien plus long que la première. Cela peut s'expliquer par le fait que la recherche locale prend moins de temps pour s'exécuter que la partie élicitation incrémentale (on peut le remarquer avec le temps d'exécution négligeable du PLS de la figure pour la procédure 1).

4.2.2 Erreur par rapport à la solution optimale du décideur

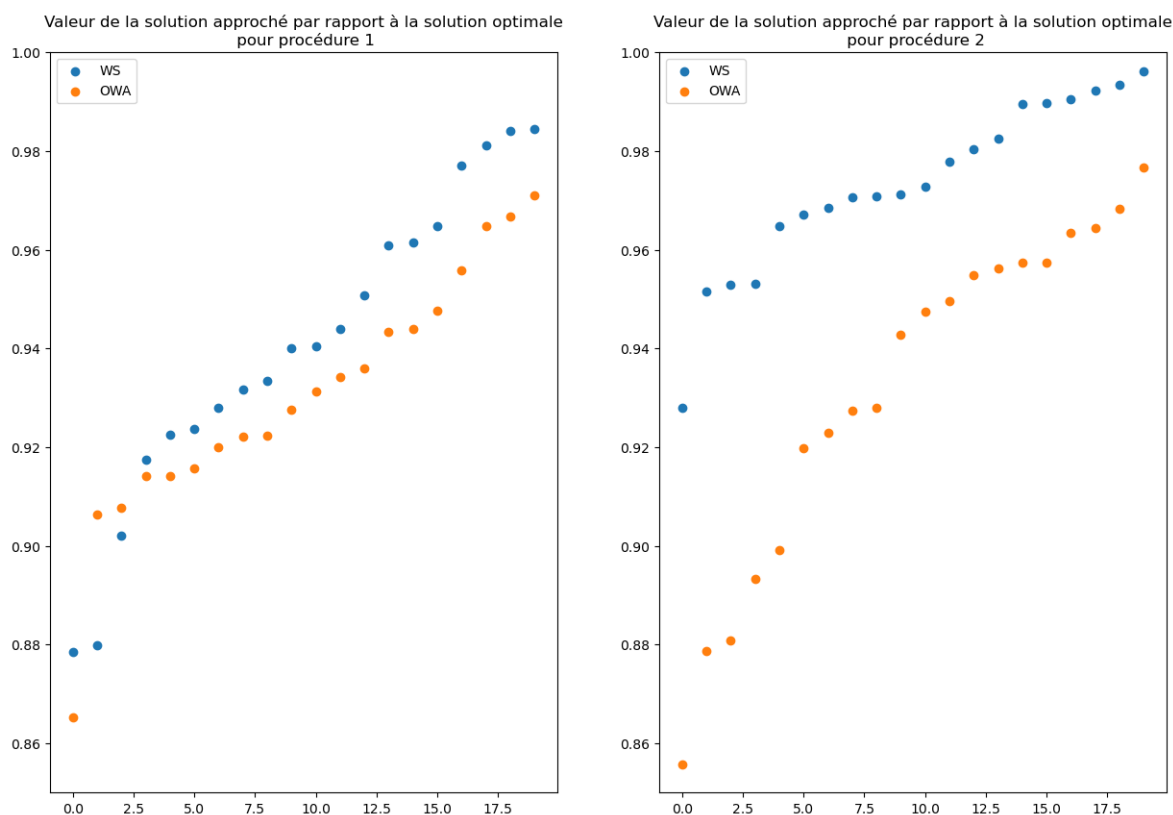


FIGURE 9 – Taux d’erreur (higher is better), trié dans l’ordre croissant

```
Rapport approché/optimal (procédure 1) moyen pour WS : 0.940320818910356, médiane : 0.9402731528761041, écart-type : 0.030669588795048514
Rapport approché/optimal (procédure 1) moyen pour OWA : 0.9305144590899174, médiane : 0.929391984746591, écart-type : 0.02432184410692402
Rapport approché/optimal (procédure 2) moyen pour WS : 0.9731984637632625, médiane : 0.9719846186917784, écart-type : 0.01700241561249446
Rapport approché/optimal (procédure 2) moyen pour OWA : 0.9322123551791481, médiane : 0.9450344153331884, écart-type : 0.03355107934664358
```

FIGURE 10 – Comparaison du taux d’erreur obtenu pour chaque procédure

Nous remarquons que le rapport approché/optimal est plus élevé pour la deuxième procédure surtout pour la somme pondéré : les solutions approchées obtenues par la deuxième procédure sont de bien meilleur qualité que ceux obtenus par la première.

4.2.3 Nombre de questions posées

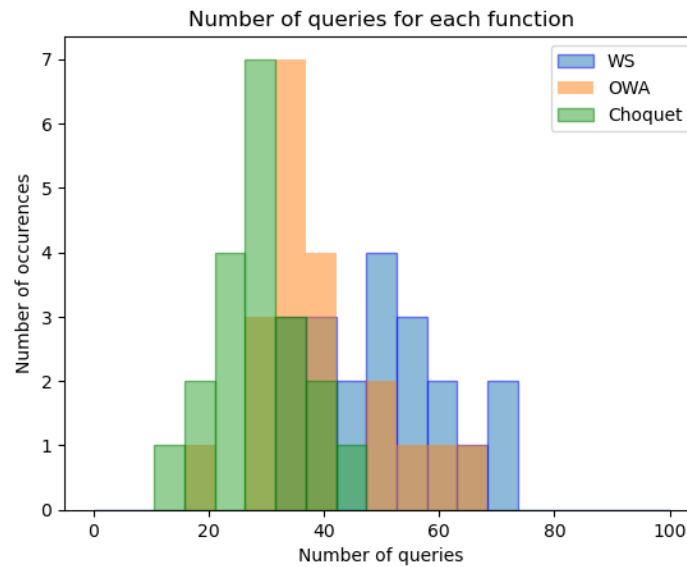


FIGURE 11 – Répartition du nombre de questions nécessaire pour chaque fonction (procédure 2)

```
Nombres de questions moyen pour WS (procédure 2): 50.5, médiane : 49.5, écart-type : 10.924742559895861
Nombres de questions moyen pour OWA (procédure 2): 38.45, médiane : 34.5, écart-type : 10.864966635935886
Nombres de questions moyen pour Choquet (procédure 2): 28.65, médiane : 28.0, écart-type : 7.391041875135061
```

FIGURE 12 – Comparaison du nombre de questions des différentes fonctions (procédure 2)

```
Nombres de questions moyen pour WS (procédure 1): 7.25, médiane : 7.0, écart-type : 1.4097872179871684
Nombres de questions moyen pour OWA (procédure 1): 5.9, médiane : 6.0, écart-type : 1.3379088160259653
Nombres de questions moyen pour Choquet (procédure 1): 3.6, médiane : 4.0, écart-type : 0.9695359714832658
```

FIGURE 13 – Rappel nombre de questions posées à la procédure 1

Nous avons encore une fois que le nombre de questions nécessaire est le plus faible avec des fonctions d'agrégations de type Choquet, suivi de OWA, et enfin de somme pondéré.

Cependant, le nombre de questions posées est bien plus élevé avec la deuxième procédure, ce qui semble être normal, sachant que nous faisons cette fois-ci appel à l'élicitation incrémentale plus de fois que pour la première procédure.

Références

1. A. JASZKIEWICZ, T. L. ND-Tree-based update : a Fast Algorithm for the Dynamic Non-Dominance Problem. <https://arxiv.org/abs/1603.04798>.
2. Minimax Regret Approaches for Preference Elicitation with Rank-Dependent Aggregators. <https://hal.science/hal-01170030>.
3. Regret-Based Elicitation for Solving Multi-Objective Knapsack Problems with Rank-Dependent Aggregators. <https://hal.sorbonne-universite.fr/hal-02493998>.
4. On MILP Models for the OWA Optimization. <https://www.ia.pw.edu.pl/~wogrycza/publikacje/artykuly/myjt12.pdf>.