Nantes Université — UFR Sciences et Techniques
Master informatique
Année académique 2024-2025

Dossier Exercice d'implémentation

Métaheuristiques

Xavier HOUDART

Djibril RAMDANI

26 novembre 2024

Livrable de l'exercice d'implémentation 1 : Heuristiques de construction et d'amélioration gloutonnes

Formulation du SPP

Le SPP (Set Packing Problem) est un problème d'optimisation où l'on cherche à sélectionner des ensembles parmi un groupe, de façon à maximiser leur nombre sans qu'ils partagent d'éléments communs. L'objectif est de choisir les ensembles qui ne se chevauchent pas.

Prenons un problème concret, un voleur braque une bijouterie et essaye d'emporter le plus d'objets de valeurs possible. Il devra donc choisir un maximum d'objets a mettre dans son sac, qui n'a pas une contenance infinie afin de maximiser le prix de son butin.

Modélisation JuMP (ou GMP) du SPP

JuMP est un créateur de modèles qui permet, a partir de données (La matrice de contraintes et les coûts des éléments) de générer des modèles (Modèles GLPK ou HiGHS) afin de résoudre automatiquement les problèmes de SPP

Instances numériques de SPP

Instance	**# Variables**	**# Contraintes**	**Meilleure valeur connue**
pb_100rnd0200.dat	100	500	34*
pb_200rnd0100.dat	200	1000	416*
pb_200rnd0400.dat	200	1000	64*
pb_200rnd1000.dat	200	200	118*
pb_200rnd1700.dat	200	600	255*
pb_500rnd0500.dat	500	2500	122*
pb_500rnd0600.dat	500	2500	8
pb_500rnd1800.dat	500	1500	13
pb_1000rnd0400.dat	1000	5000	48
pb_2000rnd0800.dat	2000	2000	135

Table 1 – Tableau des instances uttilisées

Heuristique de construction appliquée au SPP

Pour notre construction de la première solution, nous avons uttilisés un algorythme glouton dont le code se situe après :

Sur le fichier d'exemple (didacticSPP), qui est modélisé par le C et le A suivant :

C = [10, 5, 8, 6, 9, 13, 11, 4, 6]

A = [1 1 1 0 1 0 1 1 0; 0 1 1 0 0 0 0 1 0; 0 1 0 0 1 1 0 1 1; 0 0 0 1 0 0 0 0 0; 1 0 1 0 1 1 0 0 1; 0 1 1 0 0 0 1 0 1; 1 0 0 1 1 0 0 1 1]

Nous avons uttilisés un algorithme glouton pour réaliser une solution x0 grâce a ce code :

```

# ---------------------------------------------------------------------------- #
# Mettre une solution x0 (avec l'algo glouton)

function constructionX0(C, A)
  m, n = size(A)
  final = zeros(Int,m)
  final2 = zeros(Int,n)
  cAutre = Float16[]
  score = 0

  #
  for i = 0:n-1
    nb1 = 0
    for j = 1:m
      if (A[j+i*m]) == 1
        nb1 +=1
      end
    end
    push!(cAutre,(C[i+1]/nb1))
  end

  cOrdre = sort(cAutre, rev=true)

  for i = 1:n

    for j = 1:n

      if cOrdre[i] == cAutre[j]

        tbl10 = []
        for k = 1:m
          push!(tbl10,A[(j-1)*m+k])
        end

        test = true
        for k = 1:m

          if tbl10[k] == 1 & final[k] == 1
            test = false
          end

        end

        if test
          final2[j] = 1

          for l = 1:m
            final[l] += tbl10[l]
          end

        end

      end

    end

  end

  for i = 1:n
    if final2[i] == 1
```

```
      score += C[i]
    end
  end

  return ([final2,score])

end
```

Ce code nous permet, comme vu en cours, de trouver une bonne solution grâce a l'algorythme glouton.

Solution trouvée : [0, 0, 0, 1, 0, 1, 1, 0, 0]

Poids : 30

Heuristique d'amélioration appliquée au SPP

Par la suite, on a créés un algorithme de décente pour trouver une meilleure solution représenté par ce code :

```
function test(C, A, solutionATester, scoreABattre)
    m, n = size(A)
    final = zeros(Int, m)
    reponseAcceptee = true

    for i = 1:size(solutionATester)[1]
        if solutionATester[i] == 1
            for j = 1:m
                final[j] += A[m*(i-1)+j]
            end
        end
    end

    for i = 1:size(final)[1]
        if final[i] > 1
            reponseAcceptee = false
        end
    end

    if reponseAcceptee
        score = 0
        for i = 1:size(solutionATester)[1]
            if solutionATester[i] == 1
                score += C[i]
            end
        end
        return(["O", solutionATester, score])
    else
        return(["N", [], 0])
    end
end

function decente(C, A, solutionx0, score)
    solutiondepart = copy(solutionx0)
    solutionTrouvee = true

    while solutionTrouvee
        solutionTrouvee = false
        for i = 1:size(solutionx0)[1]
            if solutionx0[i] == 1
```

```
                for j = 1:size(solutionx0)[1]
                    if solutionx0[j] == 0
                        solution2 = copy(solutionx0)
                        solution2[i] = solutionx0[j]
                        solution2[j] = solutionx0[i]

                        retour = ["N", [], 0]
                        if solutionTrouvee == false
                            retour = test(C, A, solution2, score)
                        end

                        if retour[1] == "O"
                            solutionx0 = retour[2]
                            score = retour[3]
                            solutionTrouvee = true
                        end
                    end
                end
            end
        end
    end

    if solutiondepart == solutionx0
    println("Pas d'am liorations apr s la descente")
else
    println("Am lioration de la solution ", solutiondepart, " en ",
       solutionx0, " gr ce   l'algorithme de descente")
    println("Score ", score)
end
return(solutionx0, score)
end
```

Le code va parcurir toutes les permutations de 1 avec les 0 de la solutions. Pour chaque solution, il va lancer la fonction test qui a pour but de vérifier si une solution est valide. Si la fonction test lui renvoie un tableau qui commence par un "N", le programme sait que il ne doit pas lire la suite, car il s'agit soit d'une solution invalide, soit d'une solution qui est moins grande que la solution x0

Si il y a une solution valide qui est retournée, alors la fonction de décente va prendre cette solution et faire les mêmes tests que précedemment jusqu'a ce qu'il n'y ait pas de solutions valides qui ressort.

Le résultat va ensuite être renvoyé au programme main, qui va garder cette solution améliorée. Dans notre cas :

Pas d'amélioration sur la décente

Expérimentation numérique

La machine sur laquelle j'ai fait les tests est un ordinateur **OMEN Laptop 15** avec un processeur **Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz** et avec **16,0 Go** de RAM

Résultats : Présenter sous forme synthétique (tableau, graphique...) les résultats obtenus pour les 10 instances sélectionnées.

Discussion

Instance	Temps total (secondes)	Résultat	temps GLPK	limite GLPK
pb_100rnd0200.dat	0.0503232	29	50.0	0.0037512
pb_200rnd0100.dat	0.2827907	365	581.067500494683	0.0275364
pb_200rnd0400.dat	0.4541357	55	100.0	0.012782
pb_200rnd1000.dat	0.3040121	115	118.0	0.0030455
pb_200rnd1700.dat	0.1316635	223	350.57602162237964	0.0199578
pb_500rnd0500.dat	0.157107	84	373.30228293506565	0.2540663
pb_500rnd0600.dat	0.7611126	6	28.311930490364308	0.7879675
pb_500rnd1800.dat	0.693873	9	31.273204969054706	0.3897739
pb_1000rnd0400.dat	0.6368316	49	330.2617032793375	2.2027946
pb_2000rnd0800.dat	57.1426824	113	3.3864071	168.82335306104642

Table 2 – Tableau des instances uttilisées

Questions type pour mener votre discussion :

— au regard des temps de résolution requis par le solveur GLPK pour obtenir une solution optimale à l'instance considérée, l'usage d'une heuristique se justifie-t-il ?

  Lorsqu'on se base sur les résultats obtenus, l'usage d'une heuristique se justifie totalement. En effet, la rapidité des heuristiques, comme le glouton ou descente, nous donne la possibilité d'obtenir des bonnes solutions viables dans des temps courts, là où le solveur GLPK prend beaucoup moins de temps mais ne trouve qu'une limite, qui est pas forcément vraie et qui ne donne pas forcément une solution réalisable. Ainsi, on peut donc constater que les heuristiques sont une solution incontournable pour traiter efficacement des problèmes à grande échelle.

— avec pour référence la solution optimale, quelle est la qualité des solutions obtenues avec l'heuristique de construction et l'heuristique d'amélioration ?
  Sur le plan des temps de résolution, quel est le rapport entre le temps consommé par le solveur MIP et vos heuristiques ?

  Les solutions trouvées sont de bonnes solutions. En effet, le glouton suivi d'une décente ne donne pas une mauvais solutions même si ce n'est pas une solution parfaite. La solution doit être coincée dans un maximum local, dont il ets imposible de sortir en uttilisant une décente.

— Le recours aux (méta)heuristiques apparaît-il prometteur ?
  Entrevoyez-vous des pistes d'amélioration à apporter à vos heuristiques ?

  L'uttilisation des métaheuristiques semble prommeteur. Les pistes a explorer pour améliorer le travail réalisé serait de réussir a passer ces maximums locaux, notemment avec le GRASP

GRASP et ReactiveGRASP pour le Set Packing Problem (SPP)

Xavier Houdart et Djibril Ramdani

Année académique : 2024-2025

Mise en Place de GRASP

L'algorithme GRASP (*Greedy Randomized Adaptive Search Procedure*) utilise deux phases principales pour générer une solution optimisée pour le SPP :

1. **Construction d'une solution initiale :** Génération d'une solution faisable avec un processus glouton randomisé.
2. **Recherche locale :** Optimisation de la solution initiale en explorant son voisinage immédiat.

Fonction `constructionSolution(C, A, alpha)`

Cette fonction crée une solution réalisable initiale pour le SPP en utilisant α pour ajouter de la variabilité au processus glouton.

```
function constructionSolution(C, A, alpha)
    m, n = size(A)
    solution = zeros(Int, n)
    contraintes = zeros(Int, m)
    ensembleDeSolutions = []
    pasFini = true
    score = 0
    cAutre = Float16[]
    while pasFini
        ensembleDeSolutions = []
        for i = 1:n
            vraiFaux = verifieBon(C, A, contraintes, i, solution)
            if vraiFaux
                push!(ensembleDeSolutions, [compteCDiviseParnb1(C, A, i),
                    i])
            end
        end
        if isempty(ensembleDeSolutions)
            pasFini = false
        else
            ensembleDeSolutions = sort!(ensembleDeSolutions)
            nbAGarder = size(ensembleDeSolutions)[1] - Int(round(alpha *
                size(ensembleDeSolutions)[1]))
            if nbAGarder == 0
                nbAGarder = 1
            end
            ensembleDeSolutions = last(ensembleDeSolutions, nbAGarder)
            nb = rand(1:size(ensembleDeSolutions)[1])
            sol = ensembleDeSolutions[nb]
            for i = 1:m
                contraintes[i] += A[size(A)[1] * (Int(sol[2]) - 1) + i]
```

```
            end
            solution[Int(sol[2])] = 1
        end
    end
    return solution, score
end
```

Fonction grasp(C, A, tour, alpha)

Cette fonction utilise `constructionSolution` pour générer une solution initiale, puis applique une descente locale pour l'améliorer.

```
function grasp(C, A, tour, alpha)
    bestSol = 0
    bestScore = 0
    for i = 1:tour
        solution, score = constructionSolution(C, A, alpha)
        solution, score = descente(C, A, solution, score)
        if score > bestScore
            bestScore = score
            bestSol = solution
        end
    end
    return bestScore, bestSol
end
```

Ajout du Composant ReactiveGRASP

ReactiveGRASP adapte dynamiquement α en fonction des performances obtenues au cours des itérations, améliorant ainsi la capacité de GRASP à explorer l'espace de solutions.

```
function reactiveGRASP(C, A, pas, Nalpha, tours)
    nbTbl = Int(floor(1 / pas))
    tblAlpha = []
    tblAvg = []
    tblqDek = []
    for i = pas:pas:1
        push!(tblAlpha, [i, 1 / nbTbl])
        push!(tblAvg, [i, 0, 0])
    end
    sBest = []
    zBest = 0
    zWorst = 10^10
    for i = 1:nbTbl
        score, solution = grasp(C, A, 1, tblAlpha[i][1])
        if score > zBest
            zBest = copy(score)
            sBest = copy(solution)
        end
        if score < zWorst
            zWorst = copy(score)
        end
        tblAvg[i][2] += score
        tblAvg[i][3] += 1
    end
    for i = 1:tours
        for j = 1:Nalpha
            alpha = nbAlea(tblAlpha)
            score, solution = grasp(C, A, 1, tblAlpha[alpha][1])
```

```
                if score > zBest
                    zBest = copy(score)
                    sBest = copy(solution)
                end
                if score < zWorst
                    zWorst = copy(score)
                end
                tblAvg[alpha][2] += score
                tblAvg[alpha][3] += 1
            end
            SommeqDei = 0
            for j = 1:size(tblAlpha)[1]
                zAvg = tblAvg[j][2] / tblAvg[j][3]
                push!(tblqDek, ((zAvg - zWorst) / (zBest - zWorst)))
                SommeqDei += ((zAvg - zWorst) / (zBest - zWorst))
            end
            for j = 1:size(tblAlpha)[1]
                pDek = tblqDek[j] / SommeqDei
                tblAlpha[j][2] = max(pDek, 0.000001)
            end
        end
        return zBest, sBest
end
```

1 Tableau récapitulatif des résultats

Fichier	Méthode	Temps (secondes)	Z (Score)	GLPK (Z)
didacticSPP.dat	Glouton + descente	0.0437	30	30.0
	GRASP (20,0.75)	0.0005	30	
	ReactiveGRASP (0.05,15,20)	0.0086	30	
	Avec GLPK	0.0010	-	30.0
pb_1000rnd0400.dat	Glouton + descente	0.0912	40	330.26
	GRASP (20,0.75)	2.3715	59	
	ReactiveGRASP (0.05,15,20)	36.5139	67	
	Avec GLPK	2.0207	-	330.26
pb_100rnd0200.dat	Glouton + descente	0.0269	30	50.0
	GRASP (20,0.75)	0.5767	31	
	ReactiveGRASP (0.05,15,20)	8.5205	32	
	Avec GLPK	0.0038	-	50.0
pb_200rnd0100.dat	Glouton + descente	0.1967	365	581.07
	GRASP (20,0.75)	5.6167	403	
	ReactiveGRASP (0.05,15,20)	116.6724	414	
	Avec GLPK	0.0229	-	581.07
pb_200rnd0400.dat	Glouton + descente	0.3119	57	100.0
	GRASP (20,0.75)	5.8933	58	
	ReactiveGRASP (0.05,15,20)	91.5351	59	
	Avec GLPK	0.0089	-	100.0
pb_200rnd1000.dat	Glouton + descente	0.1580	115	118.0
	GRASP (20,0.75)	3.2338	115	
	ReactiveGRASP (0.05,15,20)	51.5594	116	
	Avec GLPK	0.0027	-	118.0
pb_200rnd1700.dat	Glouton + descente	0.0648	223	350.57
	GRASP (20,0.75)	1.0452	233	
	ReactiveGRASP (0.05,15,20)	20.7520	243	
	Avec GLPK	0.0141	-	350.57
pb_500rnd0500.dat	Glouton + descente	0.0683	84	373.30

Suite de la page suivante

Fichier	Méthode	Temps (secondes)	Z (Score)	GLPK (Z)
	GRASP (20,0.75)	1.7989	117	
	ReactiveGRASP (0.05,15,20)	31.5817	115	
	Avec GLPK	0.2473	-	373.30
pb_500rnd0600.dat	Glouton + descente	0.0841	6	28.31
	GRASP (20,0.75)	1.4999	7	
	ReactiveGRASP (0.05,15,20)	23.0839	7	
	Avec GLPK	0.7815	-	28.31
pb_500rnd1800.dat	Glouton + descente	0.0818	9	31.27
	GRASP (20,0.75)	1.6614	11	
	ReactiveGRASP (0.05,15,20)	24.0966	12	
	Avec GLPK	0.3786	-	31.27

Livrable de l'exercice d'implémentation 3 : Algorithme Génétique pour le SPP

Question 1

Pour votre problème de SPP, mettre en place une seconde métaheuristique (autre que celle déployée dans les EI précédents) parmi les métaheuristiques autres que GRASP/VNS/ILS (SA, TS, GA, ACO...) abordées en cours.

Dans cette question, nous avons choisi de développer une métaheuristique basée sur un algorithme génétique (*Genetic Algorithm*, GA) pour résoudre le problème du *Set Packing Problem* (SPP). L'algorithme génétique est une technique d'optimisation inspirée des principes de la sélection naturelle et de l'évolution biologique. Cette méthode repose sur les concepts de population, sélection, croisement (*crossover*) et mutation.

Description de l'algorithme

L'algorithme génétique proposé pour le SPP suit les étapes suivantes :

1. Création de la population initiale

Nous avons utilisé la métaheuristique *ReactiveGRASP* pour générer les solutions initiales. Le meilleur paramètre α trouvé par *ReactiveGRASP* est utilisé pour guider la distribution des solutions. La population initiale est composée de :

- 100 solutions générées en ajustant α autour de sa meilleure valeur avec des variations prédéfinies (par exemple, ± 0.05, ± 0.1, etc.).
- 50 solutions générées aléatoirement avec des valeurs de α comprises entre 0 et 1.

2. Évaluation des solutions

Chaque solution est évaluée en fonction de son score (la qualité de la solution pour le SPP). Les solutions sont ensuite triées par ordre décroissant de score, et un processus de **normalisation des scores** est appliqué pour calculer leur aptitude (*fitness*) relative. Cette aptitude est utilisée pour la sélection des parents.

3. Sélection des parents

Une technique de sélection probabiliste est utilisée, où les solutions ayant un meilleur score ont une probabilité plus élevée d'être sélectionnées comme parents pour la génération suivante. Ce mécanisme favorise l'exploitation des meilleures solutions tout en maintenant la diversité génétique.

4. Croisement (*Crossover*)

Pour chaque paire de parents sélectionnés, un croisement est effectué en combinant les parties des solutions parentales. Des points de croisement aléatoires sont choisis pour déterminer quelles parties des parents sont échangées pour produire des enfants.

5. Mutation

Une étape de mutation est appliquée aux enfants générés pour introduire de la diversité dans la population. La probabilité de mutation diminue exponentiellement en fonction du nombre de mutations appliquées :
- 5% de chance pour une mutation simple.
- 0,25% de chance pour deux mutations simultanées.
- Des probabilités encore plus faibles pour trois ou quatre mutations.

6. Validation des solutions

Après le croisement et la mutation, les solutions sont validées pour garantir qu'elles respectent les contraintes du SPP. Les solutions invalides sont ajustées ou supprimées.

7. Itérations

Les étapes 2 à 6 sont répétées pour un nombre prédéfini de générations (ou jusqu'à atteindre une limite de temps) afin d'améliorer progressivement la qualité des solutions.

2 Tableau récapitulatif des résultats

Fichier	**Méthode**	**Temps (s)**	**Z (Score)**	**GLPK (Z)**
didacticSPP.dat	Glouton + descente	0.0437	30	30.0
	GRASP (20, 0.75)	0.0005	30	
	ReactiveGRASP (0.05, 15, 20)	0.0086	30	
	Avec GLPK	0.0010	-	30.0
	Algorithme Génétique	0.7607	53	
pb_1000rnd0400.dat	Glouton + descente	0.0912	40	330.26
	GRASP (20, 0.75)	2.3715	59	
	ReactiveGRASP (0.05, 15, 20)	36.5139	67	
	Avec GLPK	2.0207	-	330.26
	Algorithme Génétique	60.3847	56	
pb_100rnd0200.dat	Glouton + descente	0.0269	30	50.0
	GRASP (20, 0.75)	0.5767	31	
	ReactiveGRASP (0.05, 15, 20)	8.5205	32	
	Avec GLPK	0.0038	-	50.0
	Algorithme Génétique	11.4961	33	
pb_200rnd0100.dat	Glouton + descente	0.1967	365	581.07
	GRASP (20, 0.75)	5.6167	403	
	ReactiveGRASP (0.05, 15, 20)	116.6724	414	
	Avec GLPK	0.0229	-	581.07
	Algorithme Génétique	62.6872	403	
pb_200rnd0400.dat	Glouton + descente	0.3119	57	100.0
	GRASP (20, 0.75)	5.8933	58	
	ReactiveGRASP (0.05, 15, 20)	91.5351	59	
	Avec GLPK	0.0089	-	100.0
	Algorithme Génétique	68.2101	59	
pb_200rnd1000.dat	Glouton + descente	0.1580	115	118.0
	GRASP (20, 0.75)	3.2338	115	
	ReactiveGRASP (0.05, 15, 20)	51.5594	116	
	Avec GLPK	0.0027	-	118.0
	Algorithme Génétique	42.4782	115	

Suite de la page suivante

Fichier	Méthode	Temps (s)	Z (Score)	GLPK (Z)
pb_200rnd1700.dat	Glouton + descente	0.0648	223	350.57
	GRASP (20, 0.75)	1.0452	233	
	ReactiveGRASP (0.05, 15, 20)	20.7520	243	
	Avec GLPK	0.0141	-	350.57
	Algorithme Génétique	19.1723	237	
pb_500rnd0500.dat	Glouton + descente	0.0683	84	373.30
	GRASP (20, 0.75)	1.7989	117	
	ReactiveGRASP (0.05, 15, 20)	31.5817	115	
	Avec GLPK	0.2473	-	373.30
	Algorithme Génétique	38.0390	103	
pb_500rnd0600.dat	Glouton + descente	0.0841	6	28.31
	GRASP (20, 0.75)	1.4999	7	
	ReactiveGRASP (0.05, 15, 20)	23.0839	7	
	Avec GLPK	0.7815	-	28.31
	Algorithme Génétique	32.2313	7	
pb_500rnd1800.dat	Glouton + descente	0.0818	9	31.27
	GRASP (20, 0.75)	1.6614	11	
	ReactiveGRASP (0.05, 15, 20)	24.0966	12	
	Avec GLPK	0.3786	-	31.27
	Algorithme Génétique	28.2061	12	

Analyse des Métaheuristiques

Pour répondre à la question d'évaluation des deux métaheuristiques (ReactiveGRASP et l'algorithme génétique), nous analysons leurs performances en utilisant les indicateurs pertinents extraits du tableau récapitulatif des résultats : le temps d'exécution (**Temps (s)**) et la qualité des solutions (**Z (Score)**).

2.1 Analyse en faveur de ReactiveGRASP

- **Qualité des solutions (Z)** : ReactiveGRASP obtient systématiquement des scores proches ou supérieurs à ceux des autres méthodes, notamment le glouton et GRASP simple. Par exemple, pour le fichier `pb_200rnd0100.dat`, ReactiveGRASP atteint un score de 414, supérieur à celui de l'algorithme génétique (403) et des autres approches. De plus, dans le cas de `pb_200rnd1700.dat`, il atteint 243, contre 237 pour l'algorithme génétique.
- **Fiabilité des solutions** : ReactiveGRASP produit des solutions robustes avec des scores élevés, particulièrement sur des cas complexes. Par exemple, pour `pb_500rnd0500.dat`, il atteint un score de 115, proche des meilleures solutions.

2.2 Analyse en défaveur de ReactiveGRASP

- **Temps d'exécution (Temps (s))** : ReactiveGRASP est souvent la méthode la plus lente. Par exemple, sur `pb_500rnd0500.dat`, son temps d'exécution est de 31.58 secondes, ce qui est significativement plus lent que les 38.04 secondes de l'algorithme génétique. Dans les instances de grande taille, cette lenteur est un inconvénient majeur.

2.3 Analyse en faveur de l'algorithme génétique

- **Temps d'exécution (Temps (s))** : L'algorithme génétique est généralement plus rapide que ReactiveGRASP pour des instances complexes. Par exemple, sur `pb_200rnd1000.dat`, il s'exécute en 42.48 secondes contre 51.56 secondes pour ReactiveGRASP.

— **Qualité compétitive des solutions (Z)** : L'algorithme génétique offre souvent des scores compétitifs, voire supérieurs dans certains cas. Par exemple, pour `pb_500rnd0500.dat`, il atteint un score de 103, ce qui est supérieur à GRASP (117) et proche de ReactiveGRASP (115).

2.4 Analyse en défaveur de l'algorithme génétique

— **Qualité des solutions (Z)** : Bien que compétitif, l'algorithme génétique produit parfois des solutions inférieures. Par exemple, pour `pb_200rnd1700.dat`, il atteint un score de 237 contre 243 pour ReactiveGRASP.

2.5 Conclusion

Les deux métaheuristiques présentent des forces et des faiblesses. ReactiveGRASP se distingue par la qualité des solutions générées, mais son temps d'exécution est un frein pour des instances complexes. En revanche, l'algorithme génétique offre un bon compromis entre la qualité des solutions et le temps d'exécution, le rendant plus adapté dans des contextes nécessitant une réponse rapide. Le choix entre ces deux approches dépendra donc des priorités : qualité optimale ou rapidité.