Université de Nantes — UFR Sciences et Techniques
Master informatique parcours “optimisation en recherche opérationnelle (ORO)”
Année académique 2019-2020

Dossier Devoirs Maison

Métaheursitiques

Nilson Toula – Emmanuel Rochet

16 octobre 2019

Livrable du devoir maison 2 :
Métaheuristique GRASP, ReactiveGRASP et extensions

Présentation succincte de GRASP appliqué sur le SPP

L'algorithme de GRASP mis en œuvre ici est une adaptation de notre algorithme glouton, suivit d'une plus profonde descente de type kp_exchange (un 1,2 suivie d'un 1,1).
Nous choisissons aléatoirement notre solution parmi une liste de candidats restreinte dépendante de la variable α. Plus ce paramètre tends vers 0, moins nous seront exigeant vis à vis de la qualité des utilités des candidat, et plus notre liste de candidats restreint sera étendue.
Nous sélectionnons alors aléatoirement un candidat dans cette liste de candidat restreinte affin de former notre solution.

Nous procédons ensuite à une amélioration via une simple descente de type 1-2_exchange suivie d'un 1-1_exchange.

Les variables d'entrée les plus essentiels seront : une matrice de taille n*m représentant les candidat, la liste des profits associé aux variables de cette matrice, et le paramètre α , les autres variables sont utilisé pour améliorer la vitesse de traitement de cet algorithme.

On retournera la meilleur solution (sous forme de liste binaire) et sa valeur.

```
# ——————————————————————————————————————————

function GRASP(
    cost, #Une array représentant les couts, de taille n
    matrix, #une matrice de taille m*n representant les contrantes
    n, #nombre de variables
    m, #nombbre de contraintes
    alpha # le α que nous allons utiliser
     )

     (desactive_condition, stop1, variables_actives, stop2, util, SOL) = initial(m,n)

    #Création de la solution
    while desactive_condition!=stop1 && variables_actives!=stop2
        #Calcul des utilitées
        util = Utilite(cost, matrix, desactive_condition, n, m , variables_actives,
            stop1, stop2)
        #Choix du candidat
        PosCandidat::Int64 = choixcandidat(util, alpha)
        #On ajoute le candidat à la solution
        SOL[PosCandidat] = true
        #Desactive ! le candidat selectionné et les variables des contraintes où il apparait
        Desactive!(PosCandidat, matrix, desactive_condition, m, variables_actives,n)
    end
    z = calcul_z(SOL,cost,n)
# println("Valeur Solution GRASP : " , z)
    return(SOL, z, desactive_condition)
end
```

Présentation succincte de ReactiveGRASP appliqué sur le SPP

Le réactive-GRASP est une métaheuristique visant à appliquer à un algorithme GRASP une notion d'apprentissage.
Pour se faire, après un certain nombre d'itérations de GRASP avec des α équiprobables, un recalcule des chances de de sélectionner les sont réévalué vis à vis des qualités des solutions trouvés.
Ensuite ces nouvelles probabilités de α seront à nouveau réévalué, et ainsi de suite.
Pour s'assurer que chaque valeur de α puisse être sélectionné, nous avons fait le choix d'itérer au moins une fois chaque α.

```
# ——————————————————————————————————————————

function ReactiveGRASP(
    matrix, #matrice de taille m*n representant les condion de notre SPP
    cost, #liste des couts de taille n de notre SPP
    n, #nombre de variable de notre SPP
    m, #nombre de condtion de notre SPP
    ite, #nombre de tirage au sort de α avant recalcul des probabilités
    coupe, #nombre de coupe du segment [0,1], si on à pas de liste α personalisé
    alphaset, #permet de passer une liste de α personnelle
    temps #ressource en seconde alloué a notre reactive-GRASP
    )

    #initialisation
    (p,nb_iteration,z_cumul,zBest,zWorst) = intialiser(matrix,cost, n, m, ite, coupe,
         alphaset)
    evol_p=Float64[]
    t=time()
    z_global = zeros(Int64, length(p))
    ite_global = zeros(Int64, length(p))
    nb_boucle = 0

    while (time()-t <= temps)
        append!(evol_p,p) #MaJ de notre historique de probabilité
        cpt=1
        nb_boucle += 1
        #On s'assure que chaque α s'exprime au moins une fois.
        for i in 1:length(p)
            (SOL,z, crts) = GRASP(cost, matrix, n, m, p[i])
            z_cumul[i] += z
            if z < zWorst
                zWorst = z
            elseif z > zBest
                zBest = z
            end
        end #fin for
        while (cpt <= ite)
            #Selection du α
            prob = rand(Float64)
            alpha_choisit = choix_alpha(p,prob)
            #On lance GRASP avec cet α
            (SOL, z, crts) = GRASP(cost, matrix, n, m, p[alpha_choisit])
            #Amelioration
            (SOL, z) = exchange1_1(SOL,n,m,cost,crts,matrix)
            #Réinitialisation
            nb_iteration[alpha_choisit] += 1
            z_cumul[alpha_choisit] += z
            #Mise à jours
            if z > zBest
                zBest = z
            end #fin if
            if z < zWorst
                zWorst = z
            end #fin if
```

```
            #Incrément
            cpt = cpt+1
        end #Fin while
        #On recalcul les probabilitées
        recalcul_p!(p,z_cumul,zBest,zWorst,nb_iteration,evol_p)
        #On sovegarde le nombre d'itération et les valeurs de z
        for i in 1:length(p)
            z_global[i] = z_global[i] + z_cumul[i]
            ite_global[i] = ite_global[i] + nb_iteration[i]
        end

        nb_iteration = ones(Int64, length(p))
        z_cumul = zeros(Int64, length(p))

    end #fin while
    #Calcul de aAvg
    moyenne_global=zeros(Float64,length(p))
    for i in 1:length(p)
        moyenne_global[i] = z_global[i]/ite_global[i]
    end
    zAvg = sum(moyenne_global)/length(p)

    println("zBest: ", zBest," zAvg: ", zAvg, " zWorst: ", zWorst , " nombre de
        recalcul de p: ", nb_boucle)
    println(evol_p) #A decommenter si l'on souhaite afficher l'évolution des probabilités
    return(evol_p)
end #fin reactive-GRASP

#=========================================
function recalcul_p!(p,z_cumul,zBest,zWorst,nb_iteration,evol_p)
    #initialisation
    q=zeros(Float64,length(p))
    somme_q=0

    for i in 1:length(p)
        moyenne = (z_cumul[i]/nb_iteration[i])
        q[i] = (moyenne-zWorst)/(zBest-zWorst)
        somme_q += q[i]
    end
    for i in 1:length(q)
        p[i] = q[i]/somme_q
    end

     #On additionne les proabilités entre elles
        p[i] += p[i-1]
        for i in 2:length(p)-1
    end
    #On s'assure qu'il n'y ai pas d'erreur d'arrondie pour 1
    p[length(p)] = 1
end #fin
```

Expérimentation numérique de GRASP

Afin de pouvoir faire nos test, nous avons utilisé une machine de l'université dont les caractéristiques sont les suivantes :

```
E149760H@S042pc05:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz
stepping        : 3
microcode       : 0xcc
cpu MHz         : 800.018
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4
_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept v
pid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify hwp_act_window
 hwp_epp md_clear flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
bogomips        : 7392.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

Il à été décide d'utiliser comme instances de tests des instances de différentes densités afin d'avoir une idées de comment fonctionne notre GRASP sur différentes configurations de notre SPP.

Les différentes instances utilisées sont :

- Trois instances de forte densité :
  - 100rnd400
  - 100rnd0800
  - 100rnd1000

- Trois instance de densité 1
  - 200rnd300
  - 200rnd400
  - 200rnd900

- Trois instances de faible densité
  - 200rnd1100
  - 200rnd1200
  - 200rnd1800

- La plus grosse instance du benchmark :
  - 200rnd0800

  On remarque que α influe différemment pour chaque instance.
  Évolution de la solution maximale sur 100 GRASP avec une variation des α.

Valeurs de alpha:	0.2	%age de zOpt	0.4	%age de zOpt	0.6	%age de zOpt	0.8	%age de zOpt	zOpt
pb_100rnd0400.dat	14	88	15	94	15	94	16	100	16
pb_100rnd0800.dat	34	87	34	87	36	92	34	87	39
pb_100rnd1000.dat	38	95	38	95	38	95	39	98	40
pb_200rnd0300.dat	651	89	679	93	679	93	708	97	731
pb_200rnd0400.dat	58	91	58	91	61	95	60	94	64
pb_200rnd0800.dat	70	84	76	92	74	89	70	84	83
pb_200rnd0900.dat	1190	90	1212	92	1198	90	1181	89	1324
pb_200rnd1100.dat	515	94	252	46	530	97	544	100	545
pb_200rnd1200.dat	39	91	39	91	42	98	41	95	43
pb_200rnd1800.dat	18	95	17	89	17	89	18	95	19
	Moyenne 20%	90,4	Moyenne 40%	87	moyenne 60%	93,2	moyenne 80%	93,9	

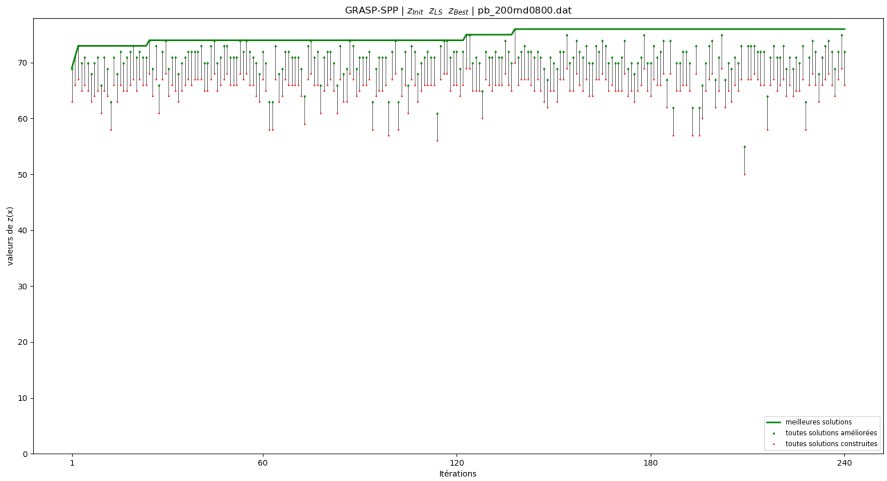
— Résultat de 10 expériences pour 1000 GRASP appliqué à nos instances pour $\alpha = 0.2$:

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9	Exp 10	%age de max/zOpt	%age de moy/zOpt	zOpt
pb_100rnd0400.dat	15	15	15	15	15	15	15	15	15	14	94	93	16
pb_100rnd0800.dat	35	36	35	35	36	35	35	34	35	35	92	90	39
pb_100rnd1000.dat	38	39	39	39	39	39	40	39	39	40	100	98	40
pb_200rnd0300.dat	693	669	685	682	668	685	669	685	669	690	95	93	731
pb_200rnd0400.dat	58	60	58	58	58	61	59	59	59	59	95	92	64
pb_200rnd0800.dat	72	73	72	72	73	71	72	71	72	72	88	87	83
pb_200rnd0900.dat	1208	1208	1206	1206	1203	1213	1205	1202	1201	1208	92	91	1324
pb_200rnd1100.dat	531	522	522	532	524	526	526	534	528	527	98	97	545
pb_200rnd1200.dat	38	40	39	39	39	40	38	39	38	39	93	90	43
pb_200rnd1800.dat	16	17	17	17	16	16	17	16	17	16	89	87	19
										Moyenne:	93,6	91,8	

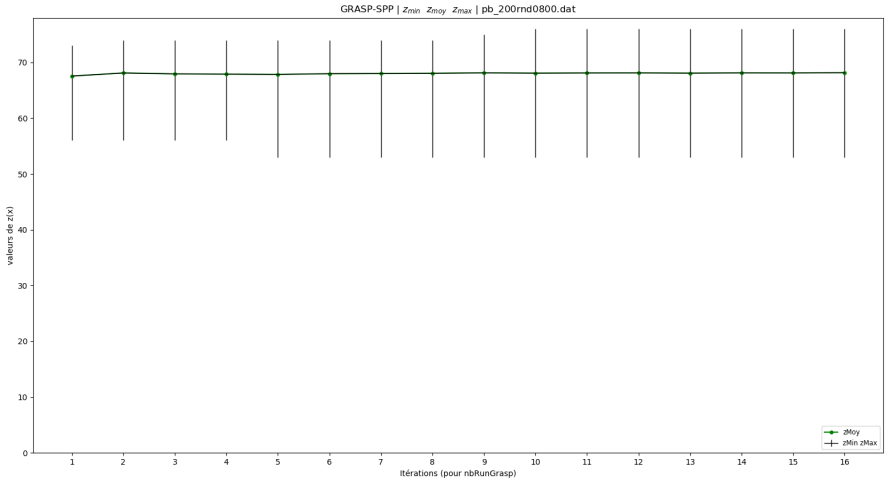
— Bilan de 10 expériences pour 1000 GRASP appliqué à nos instances pour $\alpha = 0.2$:

	zBestMax	zBestMoy	zBestMin	%age Max/Opt	zOpt
pb_100rnd0400.dat	15	14,9	14	94	16
pb_100rnd0800.dat	36	35,1	34	92	39
pb_100rnd1000.dat	40	39,1	38	100	40
pb_200rnd0300.dat	693	679,5	668	95	731
pb_200rnd0400.dat	61	58,9	58	95	64
pb_200rnd0800.dat	73	72	71	88	83
pb_200rnd0900.dat	1213	1206	1201	92	1324
pb_200rnd1100.dat	534	527,2	522	98	545
pb_200rnd1200.dat	40	38,9	38	93	43
pb_200rnd1800.dat	17	16,5	16	89	19
			Pourcentage moyen:	93,6	

Voici le profil d'un run effectué sur l'instance 200rnd0800 :



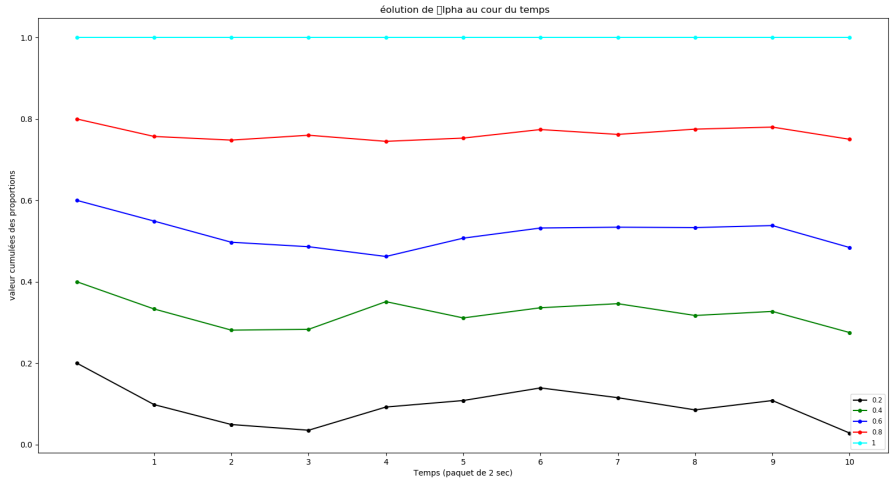
Ainsi que l'évolution de $\tilde{z}_{min}$, $\tilde{z}_{moy}$ et $\tilde{z}_{max}$ sur 16 runs de GRASP :



Expérimentation numérique de ReactiveGRASP

L'environnement matériel utilisé pour le ReactiveGRASP est le même que pour le GRASP.
De même les instances testées ne sont pas changées afin de pouvoir éventuellement voir l'amélioration apportée par de Reactive-GRASP .

Voici l'évolution de notre paramètre α sur l'instance 200rnd0800.dat :



Nous remarquons que la classe des α la plus aléatoire tends à être de moins en moins utilisée, au profit des autres notamment des classe 1.0 et 0.8 qui sont les classes les plus déterministes. Cette observation à été faite sur la plupart des autres instances, ce qui mène à penser que la structure du problème est plus adaptée aux résolutions déterministes, ou du moins comportant une part d'aléatoire assez faible. Néanmoins, nous remarquons que la classe 0.4 est ici encore très présente, avec une probabilité qui à légèrement augmentée, bien que faiblement par rapport à sa valeur de départ, ce qui amène à penser que cette classe à dû être utiles lors de ce run (peu être pour sortir d'optima locaux).

.

Voici sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées :

pb_100rnd0400.dat	zBest	16	16	16	16	16	16	16	16	16	16
	zAvg	12.4	12.4	12.4	12.4	12.4	12.4	12.4	12.4	12.4	12.4
	zWorst	7	7	8	7	8	7	7	8	8	7
	nombre de recalcul de alpha	95	97	96	96	97	97	96	96	97	97
pb_100rnd0800.dat	zBest	32	32	32	32	32	32	32	32	32	32
	zAvg	31.6	31.6	31.6	31.6	31.6	31.6	31.6	31.7	31.7	31.7
	zWorst	22	22	22	23	22	22	23	23	22	21
	nombre de recalcul de alpha	138	138	138	137	138	138	139	139	137	139
pb_100rnd1000.dat	zBest	39	39	39	39	39	39	40	39	39	39
	zAvg	35.8	35.8	35.8	35.8	35.8	35.8	35.8	35.8	35.8	35.8
	zWorst	30	29	27	30	29	30	30	27	30	30
	nombre de recalcul de alpha	55	55	55	55	55	55	55	55	55	55
pb_200rnd0300.dat	zBest	700	689	689	689	697	699	695	688	687	683
	zAvg	634	633	636	636	636	634	634	635	634	634
	zWorst	514	536	548	549	554	519	502	528	546	522
	nombre de recalcul de alpha	7	7	7	6	7	7	7	7	7	7
pb_200rnd0400.dat	zBest	59	59	61	59	59	59	62	59	58	59
	zAvg	54	54	54	54	54	54	54	54	54	54
	zWorst	47	47	44	47	48	45	46	44	47	46
	nombre de recalcul de alpha	6	6	6	6	6	6	6	6	6	6

pb_200rnd0800.dat	zBest	76	76	76	75	76	75	75	76	76	75
	zAvg	68	68	68	67	68	68	68	67	68	67
	zWorst	56	54	53	5	51	54	53	55	54	54
	nombre de recalcul de alpha	21	21	21	21	21	21	21	21	21	21
pb_200rnd0900.dat	zBest	1213	1199	1214	1199	1201	1201	1209	1209	1204	1203
	zAvg	1157	1155	1152	1156	1154	1156	1156	1156	1155	1156
	zWorst	1057	1048	1014	1036	1017	1024	1009	1067	1046	1052
	nombre de recalcul de alpha	11	11	11	11	11	11	11	11	11	11
pb_200rnd1100.dat	zBest	534	532	535	531	535	535	531	544	535	535
	zAvg	492	492	492	494	493	491	493	493	492	493
	zWorst	336	339	388	377	348	379	336	348	379	373
	nombre de recalcul de alpha	42	42	42	42	42	42	42	42	42	42
pb_200rnd1200.dat	zBest	41	42	41	42	42	42	41	42	42	42
	zAvg	37	37	37	37	37	37	37	37	37	37
	zWorst	28	28	28	28	29	29	28	28	28	28
	nombre de recalcul de alpha	37	37	36	37	37	37	37	37	36	37
pb_200rnd1800.dat	zBest	18	18	18	18	18	18	18	18	18	18
	zAvg	14	14	14	14	14	14	14	14	14	14
	zWorst	10	9	9	10	10	10	9	10	9	9
	nombre de recalcul de alpha	35	35	35	34	35	35	35	35	35	35

Discussion

Tirer des conclusions en comparant les résultats collectés avec vos deux variantes de métaheuristiques.

— En moyenne, notre algorithme glouton totalement déterministe après amélioration nous fournie des solutions autour de 88 % de la meilleur valeur connu (obtenue vie le site du Pr. X.Delorme).
10 itérations de GRASP avec un α de 0.8 fournissent des solutions à environs 91 % de la meilleur valeur connu
Enfin, les solutions apportés par le réactive-GRASP, sans path-relinking, fournissent des valeurs avoisinant les 97 % de la meilleur valeur connue.

— Cependant ces valeurs sont obtenue via de très nombreuse itérations de GRASP, peut être que ce temps de calcul pourrait être mieux exploiter avec une autre métaheuristique, moins agressive, mais ayant une plus grande convergence vers une solution optimale.
Aussi un choix de voisinage plus efficace que le simple 1-2-exchange suivit d'un 1-1-exchange pourrait suffire à atteindre de meilleures solutions via moins d'itérations.

Quelles sont les recommandations que vous émettez à l'issue de l'étude et avec quelle variante continuez vous l'aventure des métaheuristiques ?

— Nous pensons que le Reactive-GRASP est un moyen viable d'obtenir de bonnes solution, cependant dans la version que nous avons implémenté sa convergence vers n'est pas assez grande.

— Nous pensons que quelques itération de GRASP suivit de méthodes d'améliorations explorant de plus vaste voisinages pourrait apporter des solutions de meilleurs qualités qu'un Reactive-GRASP pour un coût de calcul équivalent.

— Cependant nous trouvons important de noter que les valeurs d'un α dépendent grandement de l'instance.
Ainsi, le réactive-GRASP, s'adaptant aux natures de ces instances, est moins sensible à ces variations et donc il serrait sûrement plus efficace que des solutions améliorés obtenue via des α non adapté à l'instance traité.
— Ainsi, nous pensons opter pour la variante réactive pour la suite de nos avantures.