LABORATOIRE #1 :

Optimisation par essaims particulaires pour fonctions continues et problèmes avec variables binaires

Jeudi 3 Décembre

Zoé David & Aarnaud Brun

Remarque préalable :

Le code source du projet est disponible à l'adresse suivante :
https://github.com/nonoScriptor/ParticuleSwarmOtpimisation

Afin de gagner du temps dans notre analyse des différents paramètres pour chaque fonction, nous avons fait le choix de transformer le prototype de la fonction main initiale comme ceci : `int run(int NbParam, char *Param[])`

Une nouvelle fonction main a été créée afin d'exécuter différentes fois la fonction run énoncée ci-dessus. Les paramètres de la fonction run sont donc inscrits directement dans le corps de la fonction main en vue de l'automatisation du processus :

```
/* Nombre d'appel souhaité de la fonction run */
int nb_iter = 1;

/*
Définiton des paramètres pour l'exécution de la fonction run :
arg1 = fonction : 0 <=> ALPINE / 1 <=> BANANE / 2 <=> EGGHOLDER / 3 <=> MAXCUT
arg2 = taille
arg3 = C1
arg4 = C2
arg5 = C3
arg6 = nb_eval
arg7 = nom du fichier à lancer pour maxcut
*/
/* Exécution classique */
//char* args[] = {"2", "30", "0.7", "2.0", "2.0", "10000"};
/* Exécution MAXCUT */
char* args[] = {"3", "30", "0.7", "2.0", "2.0", "10000", "CutTest.txt"};

for(int i=0; i<nb_iter; i++){
	/* Exécution normale */
	//status = run(5, args);
	/* Exécution de MAXCUT */
	status = run(6 , args);
	if (status != 0)
		cout << "ERROR in run termination";
}
```

Les résultats de ces exécutions sont stockées dans un fichier temporaire au sein du projet, qui servira de source à l'analyse exécutée par la fonction :

```
int analyse(int function_choice, int nb_run)
```

Cette fonction calcule la moyenne des différentes exécution à partir du fichier de résultats temporaire. Elle sauvegarde cette moyenne ainsi que les résultats finaux des exécutions dans un fichier qui ne sera pas effacé à la fin de l'exécution de la fonction main.

Enfin, toutes nos modifications importantes sont encadrées par les balises suivantes :

```
// % modif % //
// % modif end % //
```

Partie A : fonction Eggholder

1. Description des éléments implantés :

- Modification de l'énumération dans Entete.h :

```
enum eProb {ALPINE, BANANE, EGGHOLDER};
```

- Détermination de l'intervalle de recherche pour Eggholder :
  → Insertion dans la fonction **InitialisationIntervalleVariable** de :

```
case EGGHOLDER:
        unProb.Xmin = -512.0;
        unProb.Xmax = 512.0;
        unProb.D = 2;
        break;
```

- Ajout de la fonction objectif pour Eggholder :
  → Insertion dans la fonction **EvaluationPosition** de :

```
case EGGHOLDER: // Fonction coquetier/boite d'oeufs.
        //Optimun global -959.6407 en (512, 404.2319)
        xd = -(Pos.X[1] + 47);
        som1 += xd*sin(sqrt(fabs(Pos.X[1] + Pos.X[0]/2 + 47)));
        som2 += Pos.X[0]*sin(sqrt(fabs(Pos.X[0] + xd)));
        valeur += som1 - som2;
        break;
```

- Implémentation des informatrices :
  → Insertion dans la fonction **InitialisationInformatrices** de :

```
unPSO.NbInfo = 4;     // Chaque particule aura 4 informatrices
for(i=0; i<unPSO.Taille; i++)
{
        //Dimension du vecteur d'informatrices
        unEssaim[i].Info.resize(unPSO.NbInfo);

        vector <int> neighbour_save;
        neighbour_save.resize(unPSO.NbInfo);
        for (int j=0; j<unPSO.NbInfo; j++)
        {
                bool ok = false;
                while(!ok){
                        //Recuperer un index de particule aleatoire
                        int target = AleaDouble(0, unPSO.Taille-1);

                        //target doit etre different de index
                        // target ne doit pas exister dans le vecteur
                        if( (target != i ) &&
                                ! (std::find(neighbour_save.begin(),
                                        neighbour_save.end(),
                                        target) != neighbour_save.end()) ) {
                                //Ajouter cet index de particule à la liste
                                neighbour_save.insert(neighbour_save.end(), target);
                                ok = true;
                        }
```

```
                        //Ajouter cet index a la liste des informatrices
                        unEssaim[i].Info[j] = & unEssaim[j];
                }
        }

//Reset de la liste des particules pour le traitement de la particule suivante
        neighbour_save.clear();
}
```

2. Justification des choix :

Politique liée aux informatrices :

Nous avons choisi d'adopter une politique de voisinage de type social, c'est-à-dire que les informatrices sont fixées une seule fois à l'initialisation et ne changent plus par la suite.

Nous espérons ainsi minimiser le risque de se retrouver bloqué trop rapidement dans un optimum local.

Nous avons choisi de traiter les différentes fonctions avec 4 informatrices, car il est recommandé d'en avoir entre 3 et 5 par particule. Ces informatrices sont choisies aléatoirement parmi les particules de l'essaim.

Les informatrices d'une particule doivent toutes être différentes
Une particule ne peut avoir 2 fois la même informatrice.
Une particule ne peut pas être sa propre informatrice.
Il n'y a pas de réciprocité dans la relation particule-informatrice : une particule n'est pas forcément l'informatrice de ses informatrices.

Éléments liés à la fonction EggHolder :

Cette fonction est explicitement définie dans les consignes du laboratoire.

Le nombre de dimension est de 2.
Chaque domaine de variables s'étend entre -512.0 et 512.0.
Son expression mathématique a été découpée en plusieurs lignes de codes afin de simplifier sa création.

3. Exécutions :

Méthode employée :

Nous avons dans un premier temps fixé les paramètres à des valeurs communes pour l'exécution de ce type d'algorithme. Puis nous avons exécuter plusieurs fois le programme avec de nouveaux paramètres afin d'établir lesquels étaient les mieux pour chaque fonction.

C'est donc une méthode empirique.

Meilleurs paramètres et résultats :

Voici les résultats que nous avons établis suite à nos expériences

Fonction Alpine :

Configuration	Résultats
taille de l'essaim : 30 nombre d'informatrices : 4 C1 = 0.7 C2 = 1.5 C4 = 1.5 nombre d'évaluations : 10 000	0.000000000000 0.000000000000 0.000000000001 MOYENNE : 0.000000000000

Fonction Banane :

Configuration	Résultats
taille de l'essaim : 30 nombre d'informatrices : 4 C1 = 0.8 C2 = 1.7 C3 = 1.7 nombre d'évaluations : 10 000	0.001564680000 0.000001077190 0.000001077190 MOYENNE : 0.000522278127

Fonction EggHolder :

Configuration	Résultats
taille de l'essaim : 30 nombre d'informatrices : 4 C1 = 0.7 C2 = 2.0 C3 = 2.0 nombre d'évaluations : 10 000	-951.953000000000 -958.720000000000 -958.081000000000 MOYENNE : -956.251333333333

Partie B: PSO pour le problème de Max Cut

1. Description des éléments implantés :

- Modification de l'énumération dans Entete.h :

```
enum eProb     {ALPINE, BANANE, EGGHOLDER, MAXCUT};
```

- Activation de la structure pour traiter Max Cut dans Entete.h :

```
struct tProblem
{
        eProb  Fonction;
        int    D;
        double Xmin;
        double Xmax;
        std::string Nom;
        int NbNoeud;
        int NbArc;
        std::vector <tArc> Arc;
};
```

- Détermination de l'intervalle de recherche pour Max Cut:
  → Insertion dans la fonction **`InitialisationIntervalleVariable`** de :

```
case MAXCUT:
        unProb.Xmin = 0;
        unProb.Xmax = 1;
        //unProb.D déjà fixé par la lecture du fichier MAXCUT
        break;
```

- Ajout de la fonction objectif pour Eggholder :
  → Insertion dans la fonction **`EvaluationPosition`** de :

```
case MAXCUT:
        for(d=0; d<unProb.D-1; d++){
                vector<tArc>::iterator current_arc;
                for(current_arc = unProb.Arc.begin();
                        current_arc < unProb.Arc.end(); current_arc++){
                        if (current_arc->Ni == d){
                                if (Pos.X[d] != Pos.X[current_arc->Nj]){
                                        valeur += current_arc->Poids;
                                }
                        }
                }
        }
        break;
```

- Modification de l'initialisation des variables :
  → Insertion dans la fonction **`InitialisationPositionEtVitesseAleatoire`** de :

```
if (unProb.Fonction == MAXCUT ) {
	/* Initialisation de la position soit à 0 soit à 1 */
	Particule.Pos.X[d] = AleaDouble(unProb.Xmin, unProb.Xmax);

	// Arondir à la valeur entière la plus proche
	if (Particule.Pos.X[d] <= (unProb.Xmin + unProb.Xmax) )
		Particule.Pos.X[d] = unProb.Xmin;
	else
		Particule.Pos.X[d] = unProb.Xmax;

	Particule.V[d] = AleaDouble(-4, 4);
}
```

- Transformation de la fonction objectif en un maximum :
  → Insertion dans la fonction **`InitialisationEssaim`** de :

```
if (unProb.Fonction == MAXCUT)
	/* MAXCUT, on doit maximiser la fonction objectif */
	if (unEssaim[i].Pos.FctObj > Meilleure.FctObj)
		Meilleure = unEssaim[i].Pos;
```

  → Insertion dans la fonction **`DeplacerEssaim`** de :

```
if (unProb.Fonction == MAXCUT){
	if(unEssaim[i].Pos.FctObj >= unEssaim[i].BestPos.FctObj){
		unEssaim[i].BestPos = unEssaim[i].Pos;
		//Mémorisation du meilleur résultat atteint jusqu'ici pour MAXCUT
		if(unEssaim[i].BestPos.FctObj > Meilleure.FctObj)
			Meilleure = unEssaim[i].BestPos;
	}
}
```

- Modification du déplacement d'une particule :
  → Insertion dans la fonction **`DeplacerUneParticule`** de :

```
if (unProb.Fonction == MAXCUT){
	vector<double> sigmoide;
	sigmoide.resize(unProb.D);
	for(d=0; d<unProb.D; d++){
		Particule.V[d] =      unPSO.C1 * Particule.V[d] +
			AleaDouble(0,unPSO.C2)
			* (Particule.BestPos.X[d] - Particule.Pos.X[d])
			+ AleaDouble(0,unPSO.C3)
			* (MeilleureInfo->BestPos.X[d] - Particule.Pos.X[d]);

		// Confinement de la vitesse sur l'intervalle [-4; 4]
		if (Particule.V[d] < -4.0)
			Particule.V[d] = -4.0;
		if (Particule.V[d] > 4.0)
			Particule.V[d] = 4.0;
```

```
                double sigmo_tmp = 1.0 / (1.0 + exp(-Particule.V[d]));
                sigmoide[d] = sigmo_tmp;
        }
        // MAJ de la position de la particule :
        for(int d=0; d<unProb.D; d++){
                double proba = AleaDouble(0, 1);
                if ( proba < sigmoide[d])
                        Particule.Pos.X[d] = 1; // Probabilité validée
                else
                        Particule.Pos.X[d] = 0; // Probabilité non suffisante
        }
}
```

2. Justification des choix :

Politique liée aux informatrices :

Nous avons employé la même politique que lors de la partie 1.

Éléments liés à la fonction Max Cut :

Cette fonction correspond à la somme du poids des arcs entre deux noeuds n'appartenant pas à la même partie de la coupe

Le nombre de dimension correspond au nombre de noeuds du graphe.
Chaque domaine de variables s'étend entre 0 et 1 : le noeud est soit dans l'intérieur de la coupe, soit à l'extérieur

Redéfinition du déplacement des particules. En travaillant avec des variables binaires, la vitesse des particules sert à calculer la probabilité que le noeud passe d'un côté de la coupe ou de l'autre.

Cette probabilité de passage d'un côté ou de l'autre de la coupe est établie par l'application de la fonction sigmoïdale à la vitesse de la particule.

3. Exécutions :

Méthode employée :

Comme lors de la partie 1, nous avons employé une méthode empirique.

En raison de la durée de chacune des exécutions (qui pouvait durer plus de 15 minutes), nous avons réalisé moins d'exécution que précédemment.

Meilleurs paramètres et résultats :

Voici les meilleurs résultats que nous avons établis suite à nos expériences

Fichier G1.txt :

Configuration :

taille de l'essaim : 30

nombre d'informatrices : 4

C1 = 1.0

C2 = 2.0

C4 = 2.0

nombre d'évaluations : 10 000

Résultat : 78

Fichier G2.txt :

Configuration :

taille de l'essaim : 30

nombre d'informatrices : 4

C1 = 1.0

C2 = 2.0

C4 = 2.0

nombre d'évaluations : 10 000

Résultat : 86

On constate que sur les expériences réalisées, ce sont les mêmes configurations qui ressortent comme étant les meilleures