Problème de classification

P-median

Nicolas BLANC
BLAN30039307

Introduction

Ce rapport présente la mise en place, et l'évaluation de différents algorithmes de métaheuristique pour la résolution du problème de classification.

Tout d'abord, il y aura une présentation du problème étudié, ensuite, une analyse d'article sur le problème. Après, une description des algorithmes et pour finir l'analyse des résultats obtenues avec les métaheuristiques.

Table des matières

Le problème de classification ou localisation (p-median)

Description

Le problème de localisation est un problème assez courant que l'on retrouve dans un certain nombre d'activité. On peut facilement représenter ce problème avec un graphe. Certains nœuds sont appelés des centres et le but est de relié tous les nœuds au centre le proche. Il faut donc déterminer quels nœuds permettent de minimiser la distance à parcourir entre les nœuds et les centres. Ce problème est NP-Difficile et le nombre de solution possible est exponentiel. La plus petite instance propose un problème avec 100 villes et 5 centres, $C_5^{100} = 7.10^7$ solutions posibles.

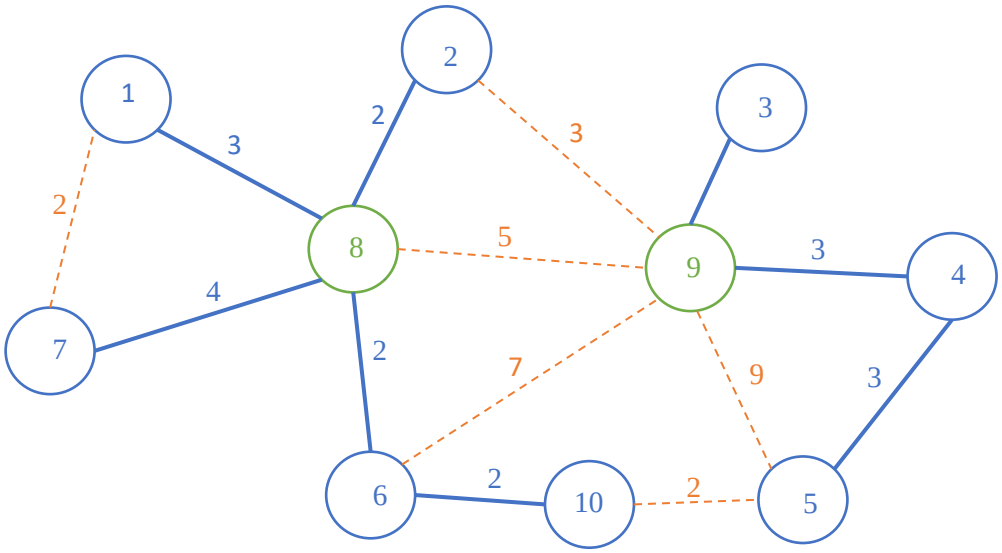

Figure 1 : Exemple de graphe de solution pour P-median.

La figure 1 propose un problème 2-median contenant 9 nœuds. Les arcs en pointillé sont des liens plus longs et dégrade la solution, mais cela permet de montrer que le graphe.

Ce problème de recherche opérationnel est assez commun et permet, entre autres de pouvoir positionner des activités en fonction du nombre de clients, là où le magasin aura le plus d'importance. Il peut permettre aussi de positionner des entrepôts plus larges pour fournir d'autres lieux plus petits. Le problème P-median à de nombreux intérêts, on le retrouve sur les différents réseaux (routiers, aériens ou téléphoniques), l'emplacement des services d'urgences, les réseaux de communications et informatiques, l'intelligence artificiel et les modèles statistiques, etc.

Description mathématique

Le but est donc de minimiser les distances entre les nœuds et les centres :

$$Minimise \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij} x_{ij}$$

Le problème parcourt deux matrices, la première d, est une matrice de distance, c'est-à-dire que chaque élément de la matrice est une distance entre deux nœuds, sans différencier si cette distance passe par d'autres nœuds. Un élément de la deuxième matrice est à 1 s'il est relié 0 un centre.

$$(1): \sum_{i=1}^{m} x_{ij} = 1\,, \qquad \forall\, j = 1, 2, \dots, n$$

$$(2):\ 0 \le x_{ij} \le y_i, \qquad \forall\, i = 1, 2, \dots, m, \forall\, j = 1, 2, \dots n$$

$$(3): \sum_{i=1}^{m} y_i = p$$

$$(4):\ y_i \in \{0, 1\}, i = 1,2, \dots, m$$

Les contraintes lies au problème. L'équation (4) introduit un vecteur en plus y_i, celui-ci contient toutes les nœuds du graphe, et sa valeur est à 1, si c'est un centre, sinon 0. L'équation (3), est directement relié à la précédente car elle fixe le nombre de centre à p, et indique que la somme du vecteur ne doit pas être supérieur à p. L'équation (1) indique que chaque nœud est lie à un et seul centre. En effet, la somme d'une ligne de la matrice x_{ij} avec j fixe doit être égale à1. L'équation (2) assure qu'un nœud est lie à un autre nœud seulement si l'autre nœud est un centre. La valeur x_{ij} peut prendre comme valeur 1 s'il est lié à ce nœud et que celui-ci est définit comme un centre, valeur dans 1 y_i. Sinon, il prend la valeur 0.

Les articles sur le problème de classification

Sur les articles choisit les cinq introduisent mathématiquement le problème puis apporte une ou plusieurs solutions avec différents algorithmes. Parmi les cinq, un est en français (Baray). On retrouve souvent l'algorithme génétique. Notamment l'article de (Alp, Erhan, & Zvi) et de (Alcaraz, Landete, & Monge), ce dernier se basant aussi sur les travaux du premier. L'article de (Saez-Aguado & Trandafir) propose deux approches différentes intéressantes, une heuristique avec la relaxation de Lagrangean et une implémentation de GRASP, vu pendant les présentations. Enfin, Les article de (Baray) et de (Mladenovic, Brimberg, Hansen, & Moreno-Pérez) propose un ensemble d'algorithme, notamment la recherche par tabou ou colonie de fourmi optimisé et en plus l'article français propose une implémentation différente de la solution et une autre approche de l'algorithme génétique.

Algorithmes de métaheuristique

Les paramètres généraux du problème sont les même pour toutes les implémentations, c'est-à-dire le nombre de ville, le nombre de centre et la matrice de distance. A la lecture du fichier cette dernière est remplie avec les arcs fournies dans le fichier, puis l'algorithme de Floyd-Warshall est appliqué. Celui-ci permet de compléter la matrice et chaque élément de la celle-ci est une distance entre deux nœuds, même si cela passe par d'autres nœuds.

Ensuite, une solution est un vecteur de p éléments contenant les centres. C'est la même implémentation quel que soit l'algorithme utilisé. Cela permet de pouvoir facilement les comparer. La fonction *createSolution()* permet de créer une nouvelle solution aléatoire avec pour contrainte que une ville ne peut pas être en double. *evaluateSolution()* permet d'évaluer la solution.

La condition d'arrêts des algorithmes est un nombre d'itérations de la boucle extérieur. Je l'ai fixé petit, 100 itérations, mais les résultats sont satisfaisant, et le nombre d'évaluation totale est supérieur du fait du voisinage ou de la fonction de croisement implémenté. Voir plus loin l'algorithme de descente et génétique.

La fonction *main()* dans p-median permet de lancer la totalité des tests. NUMBER_OF_EXECUTION_PER_ALGO permet de fixer le nombre de test par algorithme pour chaque problème, NUMBER_OF_EXECUTION_RAND permet de fixer le nombre de solutions aléatoire pendant l'initialisation de la solution pour la descente et le recuit simulé, enfin NUMBER_OF_EXECUTION_ALGO permet de fixer le nombre d'itérations pour les autres algorithmes.

Le vecteur *files* contient le nom de tous les fichiers des problèmes.

Les paramètres des algorithmes sont à remplir à la l'instanciation des algorithmes (Ligne 70, 77 et 85 de P-median.cpp).

Enfin, le programme produit trois fichiers. Le premier *testDelta.csv* contient des informations sur l'exécution du dernier algorithme de recuit simulé et s'importe dans le fichier excel AnalyseDonnees. Le second, le fichier *result.csv* contient la sortie finale du programme pour être analyser dans *AnalyseResultat*. Enfin le out.txt contient toutes les informations de sortie de l'ensemble du programme, plus lisible que *result.csv*. La sortie console ne montre qu'un avancement du programme sans information détaillé.

L'algorithme aléatoire

L'algorithme aléatoire mis en place sert pour l'initialisation des solutions. Il construit simplement une solution en récupérant un nombre de nœuds égale au nombre de centre du problème, en s'assurant qu'un nœud n'est pas mis en double dans la solution. La génération aléatoire est faite un certain nombre de fois (fixé à 100 pour les tests) et retourne la meilleure solution pour l'algorithme de descente et le recuit simulé.

L'algorithme de descente

L'algorithme de descente utilise une stratégie d'orientation pour connaitre le point suivant. Elle est composée d'un parcours dans le voisinage et d'une règle de pivot.

La stratégie mise en place est composé de deux autres stratégies avec une règle de changement. Tout d'abord la première est une stratégie orientée et le pivot est Best-improve. Un +1 est appliqué sur un nœud pour avoir le suivant, puis un -1 est appliqué, la meilleure solution est retournée.

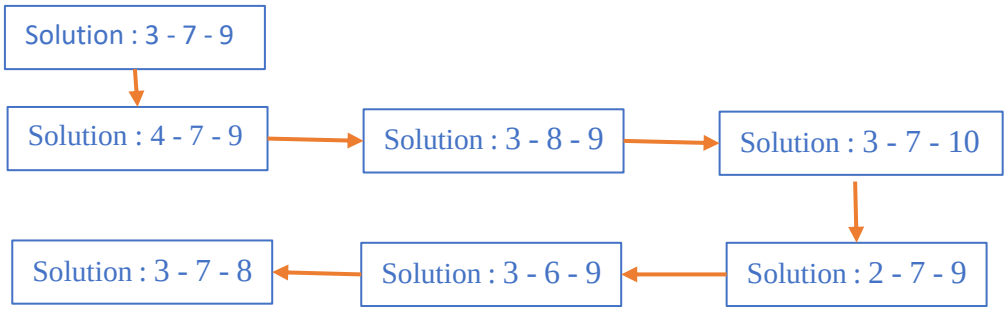

Figure 2 : Exemple d'exécution de la stratégie d'orientation

Si deux fois de suite la solution courante est la même, c'est la deuxième stratégie d'orientation qui est faite, celle-ci est aléatoire et first-improve. Elle consiste simplement à change un centre par un autre nœud, en s'assurant que le centre ne soit pas déjà présent dans la solution.

AlgorithmeDescente est une classe fille de la classe abstraite *Algorithme* et fourni donc la fonction *nextSolution()* qui implémente la stratégie d'orientation. Une variable contient la valeur précédente permettant le changement de stratégie.

L'algorithme du recuit simulé

L'algorithme du recuit simulé se base sur la diminution de la température après un recuit. Ces paramètres permettent de calculer une probabilité de faire une dégradation de la solution courante.

La recherche par tabou a été implémenter avant le recuit simulé, mais au cours du développement le choix a été changé pour le recuit simulé. Sur un grand nombre de ville, je me suis demandé si le recuit simulé n'été pas plus intéressant pour ce problème. Je trouve la possibilité d'une dégradation plus pertinente que d'éviter de boucler sur sept solutions sachant qu'il y a beaucoup de solutions possibles et que ma stratégie propose un changement aléatoire d'un centre

Le recuit simulé utilise la même stratégie d'orientation que l'algorithme de descente. Pour simplifier, la classe *RecuitSimule* hérite de *AlgorithmeDescente* et utilise donc *nextSolution()* de la classe mère. La classe rajoute le paramètre nécessaire à l'exécution de l'algorithme.

A l'inverse des autres algorithmes, celui-ci est réalisé dans le fichier *main*, en effet il influe sur le déroulement de la boucle, il n'est pas possible de faire une fonction qui retourne une solution à chaque itération.

J'utilise la même stratégie d'orientation pour deux raisons principales. La première est que cela me permet de comparer avec l'algorithme de descente et de garder une cohérence et je considère la stratégie mise en place comme suffisamment efficace et adaptée au recuit simulé.

L'algorithme génétique

L'algorithme génétique ce base sur l'évolution d'une population. Celle-ci est croisé puis éventuellement muté. Pour ce problème, un gène est un centre, un individu est une solution et une population, un ensemble de solution.

Pour cette implémentation, je me suis basé sur les travaux de (Alp, Erhan, & Zvi). En effet, j'ai trouvé très intéressante leur approche de cet algorithme. Il y a quelques différences, notamment sur la taille de la population, mais sinon le reste est sensiblement la même chose. J'ai aussi implémenter un crossover simple pour cet algorithme, mais aucuns tests n'ont été réalisé, sur plusieurs il était moins performant que le premier.

L'initialisation de la population n'est pas faite aléatoirement. Le but est d'avoir chaque ville apparaissant au moins deux fois dans la population donc la population minimale est de taille $^{n}/_{p} * 2$. L'article propose une population de taille assez grande, mais le calcul pour la fixer requiert le nombre total de solution possible et ce nombre est bien plus grand qu'un int, et j'avais des doutes concernant un int sur 64 bit. Il aurait donc fallu réimplémenter le stockage et les opérations pour un grand nombre très grand. Il aurait été possible de faire ce calcul avant, et passer directement la taille de la population, mais cela n'a été penser qu'après la mise ne place. Dans l'algorithme, la taille de population est de $\max(^{n}/_{p} * 4, ^{n}/_{3})$.

La population initiale ce construit comme suit : de 1 à n avec un pas de 1, puis de 2, etc. En veillant à ce toutes les villes soit prises. Exemple, avec n = 10, (1,2,3,4,5), (6,7,8,9,0), (1,3,5,7,9), (2,4,6,8,0), …

Ensuite, le choix des parents est un choix aléatoire de deux individus dans la population.

Le croisement, élément critique dans l'algorithme génétique doit être bien pensé. Les deux parents créé un seul fils, sa solution est la solution des deux parents ensemble, et la suite la ramène à une solution valide. Tout d'abord les villes présente dans les deux parents sont dites fixe, elles ne peuvent pas être enlevé de la solution enfant. Pour enlever une ville, l'algorithme test chaque solution avec en enlevant une seule ville puis garde la meilleure solution et recommence avec celle-ci jusqu'à avoir une solution valide.

Le choix de la nouvelle population est élitiste, à chaque fois qu'un enfant est créé, il est ajouté à la population actuelle à la place de la pire solution si cette solution est meilleure que la pire et qu'elle n'est pas déjà présente dans la population.

Concernant le crossover implémenter, celui-ci est simple, il découpe en deux les parents puis les ensembles les parties pour créer deux nouveaux enfants. L'autre différence est l'initialisation de la population, celle-ci est fait aléatoirement avec la création de chaque solution. Le reste de l'algorithme est sensiblement la même chose, sauf que la population enfant, qui peut être différent d'une taille 1 est ajouté après avoir créé tous les enfants.

Description des expérimentations numérique

J'utilise 24 instances corrige de OR-Library : les instances de pmed de 1 à 32 sauf 15, 19, 20, 24, 25, 28, 29 et 30. Le nombre de ville varie de 100 à 700, et les centres de 5 à 67. J'ai omis les fichiers ayant plus de centres car cela augmente grandement le temps à la résolution des algorithmes, plus que le nombre de ville total. Les solutions optimales de chaque instance sont connues. Chaque problème a été résolue cinq fois par algorithme.

Le programme a été fait en C++, sous Visual Studio 2015. L'ordinateur possède un processeur i5 6300HQ, cadencé à 2.30GHz.

Résultats et analyse

Les 24 instances ont été résolue cinq fois par algorithme et la fonction objectif moyenne des cinq résolutions qui a été gardé pour les évaluations suivantes.

Heuristique	**Aléatoire**	**A. Descente**	**Recuit Simulé**	**A. Génétique**
Déviation moyenne à l'optimum	25.00	8.36	10.64	7.99
Ecart type (Déviation)	11.18	4.31	6.51	3.45
Déviation max à l'optimum	58.09	20.40	29.80	13.58
Déviation min à l'optimum	12.47	2.47	2.97	2.36
Meilleure solution	0	9	4	11
Pire solution	24	1	10	13

Tableau 1 : Résumé des résultats obtenus

Heuristique	**Aléatoire**	**A. Descente**	**Recuit Simulé**	**A. Génétique**
Aléatoire	--	0	0	0
A. Descente	24	--	19	13
Recuit Simulé	24	5	--	13
A. Génétique	24	11	11	--

Tableau 2 : Nombre de fois où l'heuristique en ligne est meilleure que celui en colonne

En complément du tableau, aucun algorithme n'a obtenu la solution optimale. De plus, la résolution des cinq instances pour les problèmes par les algorithmes à pris plusieurs heures pour être terminé. J'ai remarqué que plus le nombre de centre est grand, indépendamment du nombre total de ville, plus le temps augmente. Evidemment, l'aléatoire est plus rapide que l'algorithme de descente, lui-même plus rapide que le recuit simulé, ce dernier étant vraiment plus rapide que l'algorithme génétique.

Dans ce tableau, on peut remarquer que les résultats sont meilleurs pour l'algorithme génétique. Etonnamment, l'algorithme de descente à de meilleurs résultats que le recuit simulé. On le voit d'ailleurs avec le nombre de meilleurs solution, l'AG est onze fois le meilleur, suivit par l'AD, qui lui ne l'est que 9. A l'inverse, quand il n'est pas le meilleur, l'AG est le pire et l'AD est le moins pire. Sans compter l'aléatoire qui sinon, a les pires résultats. Au vu de ces données, on peut se dire qu'il n'y aura peut-être pas de dominance empirique, hormis avec l'algorithme aléatoire.

Le deuxième tableau vient confirmer les premières impressions, et donne l'algorithme de descente légèrement meilleur que l'algorithme génétique., lui-même meilleur que le recuit simulé.

Sur les graphiques ci-dessous, on peut donc voir qu'il n'y a effectivement pas de dominance empirique entre les trois algorithmes. Les résultats sont très variés, mais on peut remarquer, entre l'AD et le RS que les courbes ont à la même tendance. Par contre, l'algorithme génétique correspond à l'inverse de ces courbes. Cela explique le fait que l'AG est soit le meilleur, soit le pire des trois. Par contre, il y a une domination stochastique de l'AD et l'AG sur RS. Mais pas entre les deux premiers. On peut donc faire un premier classement entre les méthodes, AG équivalent à AD, supérieur à RS puis Aléatoire.

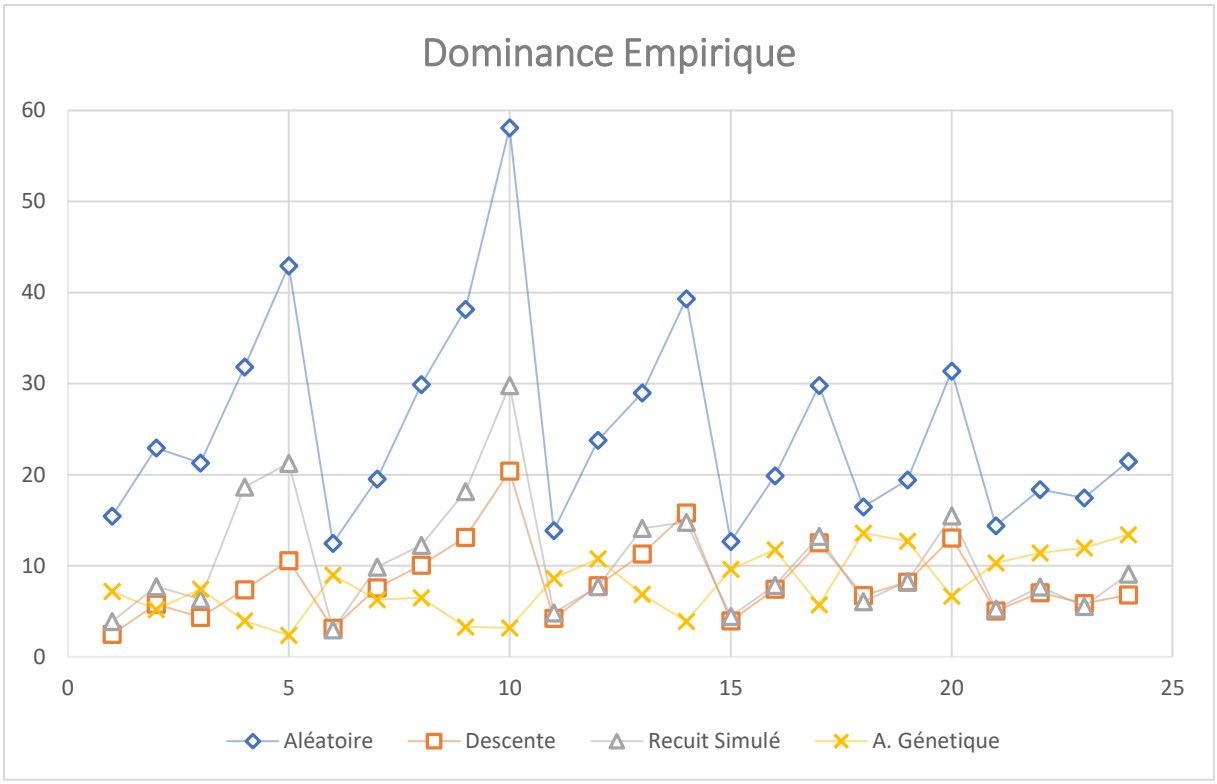

Figure 3 : Dominance empirique

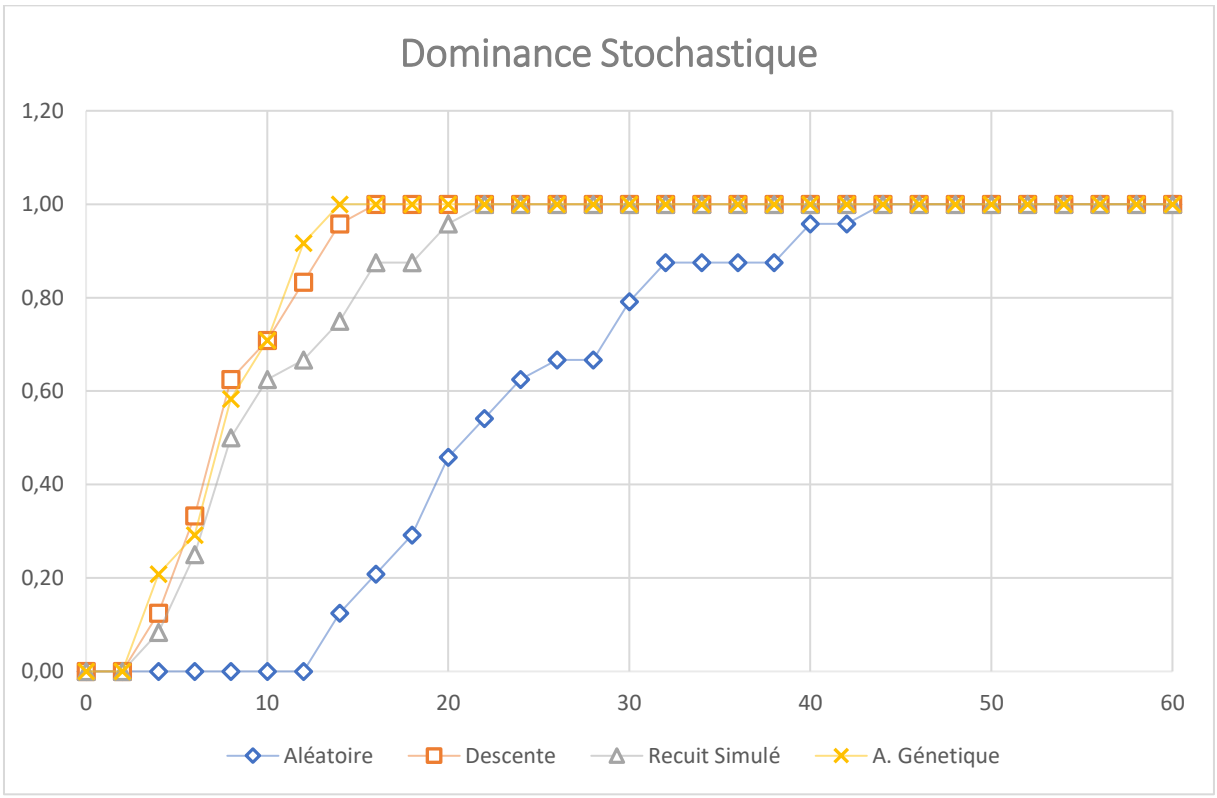

Figure 4 : Dominance stochastique

Ensuite, nous allons étudier les résultats avec le test paramétrique de Student et après le test non paramétrique de Wilcoxon.

Pour ces deux tests, un taux de confiance, alpha, de 0.05 a été retenue.

Pour H0 : m1 ⇔ m2, Student $t_{0.05,46} = 2.0141$ et $-t_{0.05,46} = -2.0141$.

Heuristique	**Aléatoire**	**A. Descente**	**Recuit Simulé**	**A. Génétique**
Aléatoire	--			
A. Descente	-6.95	--		
Recuit Simulé	-5.56	1.46	--	
A. Génétique	-7.28	-0.34	-1.80	--

Tableau 3 : Résultats du test de Student, H0 = L <=> C

Avec ces résultats, sans surprise, sans surprise, l'aléatoire est toujours dernier, par contre le test de Student montre que les trois algorithmes sont équivalents, en effet, l'hypothèse est acceptée pour les tests, AG – RS, AG – AD et RS – AD.

Le nombre d'instances de problèmes étant supérieur à vingt, c'est la Standard Normal Distribution Table qui fournit la valeur critique, $Z_{0.95} = 1.96$ et $-Z_{0.95} = -1.96$.

Heuristique	**Aléatoire**	**A. Descente**	**Recuit Simulé**	**A. Génétique**
Aléatoire	--			
A. Descente	-4.29	--		
Recuit Simulé	-4.29	3.37	--	
A. Génétique	-4.29	0	-0.94	--

Tableau 4 : Résultat du test de Wilcoxon, H0 : L <=> C

ON retrouve sensiblement les mêmes résultats que précédemment, sauf deux choses. La première montre que l'algorithme génétique et l'algorithme de descente sont effectivement équivalent, cela commençait à être visible avec Student, Z étant proche de zéro. Deuxième chose, cette fois si, l'hypothèse est rejetée pour RS ⇔ AD, avec un surclassement de l'algorithme de descente.

Pour conclure avec cette analyse, on peut établir un classement entre les différentes implémentations des algorithmes pour le problème de classification :

A. Descente ⇔A. Génétique > Recuit Simulé > Aléatoire

Conclusion

Dans l'ensemble, les résultats des algorithmes se rapproche de la fonction objective. Le fait que l'algorithme de descente soit aussi efficace est peut-être basé sur l'aléatoire

Un travail très intéressant mettant en valeurs que les point critiques du développement d'une heuristique, le voisinage pour les deux premiers, la fonction de croisement pour l'algorithme génétique. Un certain nombre de problèmes sont dans la mise en place des algorithmes dont les plus importants sont sur l'implémentation du voisinage, qui m'a conduit à passer sur le recuit simulé, et l'algorithme génétique, avec l'implémentation d'un autre croisement. Ces changements m'ont permis de découvrir les erreurs et les corrigées.

Références

Alcaraz, J., Landete, M., & Monge, J. F. (2012, Octobre 1). Problem, Design and analysis of hybrid metaheuristics for the Reliability p-Median. *European Journal of Operational Research, 222*(1), 54-64. Récupéré sur http://www.sciencedirect.com/science/article/pii/S0377221712003098

Alp, O., Erhan, E., & Zvi, D. (2003, Sep). An Efficient Genetic Algorithm for the p-Median Problem. (S. S. Media, Éd.) *Annals of Operations Research, 122*(1/4), pp. 21-24. Récupéré sur http://sbiproxy.uqac.ca/login?url=http://search.proquest.com/docview/214505945?accountid=14722

Baray, J. (s.d.). *Le modèle p-médian et sa résolution.* Récupéré sur Ecole Européenne d'Etudes Avancées: http://eeea.fr/docs/ModeleP-Median.pdf

Mladenovic, N., Brimberg, J., Hansen, P., & Moreno-Pérez, J. (2007, Juin 16). The p-median problem: A survey of metaheuristic approaches. *European Journal of Operational Research, 179*(3), 927-939. Récupéré sur http://www.sciencedirect.com/science/article/pii/S0377221706000750

Saez-Aguado, J., & Trandafir, P. C. (2012, Juillet 16). Some heuristic methods for solving p-median problems with a coverage constraint. *European Journal of Operational Research, 220*(2), 320-327. Récupéré sur http://www.sciencedirect.com/science/article/pii/S0377221712001282

Annexe

Problem.h

```
#pragma once
#include <vector>
class Problem
{
public:
	Problem(int vert, int edge, int center, int obj);
	~Problem();
	void addSolution(vector<int> center);
	void addEdge(int x, int y, int poids);
	void afficher();
	void FloydWarshall();

	int getNumberOfVertices() { return m_number_Of_Vertices; }
	int getNumberOfCenter() { return m_number_Of_Center; }
	int getValueInMatrice(int i, int j) { return ma_Graphe->getVal(i, j); }
	int getCenter(int i) { return v_Center[i]; }

	int getFctObj() { return m_FctObj; }

	void addNumberOfEvaluation() { m_number_Of_Evaluation++; }
	int getNumberOfEvaluation() { return m_number_Of_Evaluation; }
private:
	int m_number_Of_Vertices;
	int m_number_Of_Edge;
	int m_number_Of_Center;
	int m_FctObj;
	vector<int> v_Center;
	Matrice<int> * ma_Graphe;

	int m_number_Of_Evaluation;
};
```

Solution.h

```
#include "Problem.h"
#pragma once
class Solution
{
public:
	Solution();
	Solution(vector<int> solution, Problem * prob);
	Solution(vector<int> sol1, vector<int> sol, Problem * prob);
	Solution(Problem * prob);
	Solution(const Solution &obj);
	virtual ~Solution();

	void CreateSolution(Problem * problem);
	void copySolution(Solution * sol, Problem * prob);
	void copySolution(Solution * sol);

	void evaluateSolution(Problem * problem);

	void printSolution(ofstream& f_out);
	void printSolution();

	bool egal(Solution * sol);
```

```
	int getFctObj() const { return m_fctObj; }
	void resetFctObj();
	void setSolution(vector<int> * sol) { delete v_solution;  v_solution = sol; }
	vector<int> * getSolution() const { return v_solution; }
	vector<int> * addSolutionReturnFixed(vector<int> * sol);
	void addSolution(vector<int> * sol) { for each (int val in (*sol)) { v_solution-
>push_back(val); } }
	bool isPresent(int val) const { return find(v_solution->begin(), v_solution-
>end(), val) != v_solution->end(); }
	int getValue(int i) const { return v_solution->at(i); }
	void delValue(int i) { v_solution->erase(v_solution->begin() + i); }
	bool AddToValue(int i, int val, Problem * prob);
	void changeRandomCenter(Problem * prob, int vert);
	vector<int> getRangeValue(int param1, int param2);
protected:
	int m_fctObj;

	vector<int> * v_solution;
};
```

Solution.cpp

```
void Solution::CreateSolution(Problem * problem)
{
	v_solution->clear();
	m_fctObj = 0;
	int r;
	for (auto i = 0; i < problem->getNumberOfCenter(); i++)
	{
		while (isPresent(r = rand() % problem->getNumberOfVertices())) {}
		v_solution->push_back(r);
	}
}

void Solution::evaluateSolution(Problem * problem)
{
	m_fctObj = 0;
	int tmp = INT_MAX;
	for (auto i = 0; i < problem->getNumberOfVertices(); i++)
	{
		tmp = INT_MAX;
		for each (auto val in (*v_solution))
		{
			tmp = min(problem->getValueInMatrice(i, val), tmp);
		}
		m_fctObj += tmp;
	}

	problem->addNumberOfEvaluation();
}
```

Algorithme.h

```
#pragma once
class Algorithme
{
public:
	Algorithme();
	~Algorithme();

	virtual string nameAlgo() = 0;
	virtual void nextSolution(Solution * next, Solution * current, Problem * prob) =
0;
};
```

AlgorithmeDescente.cpp

```
#include "stdafx.h"
#include "AlgorithmeDescente.h"


AlgorithmeDescente::AlgorithmeDescente() :
	Algorithme()
{
	pred = new Solution();
}


AlgorithmeDescente::~AlgorithmeDescente()
{
}

void AlgorithmeDescente::nextSolution(Solution * next, Solution * current, Problem *
prob)
{
	Solution tmpSol;
	Solution * best = new Solution();

	int val, calc = 1;

	if (pred->egal(current))
	{
		int r;
		best->copySolution(current, prob);
		while (best->isPresent(r = rand() % prob->getNumberOfVertices())) {}
		best->changeRandomCenter(prob, r);
		best->evaluateSolution(prob);
	}
	else
	{
		for (auto i = 0; i < prob->getNumberOfCenter() * 2; i++)
		{
			tmpSol.copySolution(current, prob);
			if (i == prob->getNumberOfCenter())
			{
				calc = -calc;
			}

			val = i % prob->getNumberOfCenter();
			if (!tmpSol.isPresent(val + calc))
			{
				if (tmpSol.AddToValue(val, calc, prob))
```

```
                        {
                                tmpSol.evaluateSolution(prob);
                        }

                        //            tmpSol.printSolution();

                        if (tmpSol.getFctObj() < best->getFctObj())
                        {
                                best->copySolution(&tmpSol, prob);
                        }
                }
        }
    }

    next->copySolution(best, prob);
    pred->copySolution(current, prob);

    delete best;
}
```

P-Median.cpp – Recuit Simulé

```
void algorithme(RecuitSimule * algo, Solution * best, Solution * sol, Problem * prob)
{
    int i = 0;
    int delta;
    Solution * next = new Solution();

    cout << algo->nameAlgo() << endl;

    while (i < NUMBER_OF_EXECUTION_ALGO)
    {
        algo->resetNumStage();
        do
        {
            algo->nextSolution(next, sol, prob);

            delta = next->getFctObj() - sol->getFctObj();

            algo->addNbCalculDelta();
            algo->addDelta(delta);
            algo->addTemp();

            if (delta <= 0)
            {
                sol->copySolution(next, prob);

                if (sol->getFctObj() < best->getFctObj())
                {
                    best->copySolution(sol, prob);
                }

                algo->addDegradation(2);
            }
            else
            {
                if (rand() / double(RAND_MAX) < exp((-delta) / algo-
>getTemperature()))
                {
                    sol->copySolution(next, prob);
                    algo->addAcceptDegradation();
                    algo->addDegradation(1);
```

```
                        }
                        else
                        {
                                algo->addDegradation(0);
                        }
                }
                algo->addNumStage();

                /* ----- Affichage de l'avancement de l'algorithme ----- */
                if (i % SETLOAD_ALGO == 0)
                        cout << "- ";
//              cout << i << " - FctObj : " << sol->getFctObj() << "\n";
                /* ----- ----- ----- */
                i++;
        } while (algo->getNumStage() != algo->getNumberInStage());

        // Permet de verifier que le programme ne change pas de palier si le
nombre maximum est atteint
        if (algo->getNbChangeTemp() < algo->getNumberOfStage())
        {
                algo->setTemperature();
                algo->AddNbChangeTemp();
        }
    }

    cout << endl;

    delete next;

    algo->writeInformation(NUMBER_OF_EXECUTION_ALGO);
}
```

AlgorithmeGenetique.h

```
#pragma once
#include "Algorithme.h"
class AlgorithmeGenetique :
    public Algorithme
{
public:
    AlgorithmeGenetique(Problem * prob, bool co = true, float size_Pop_Child = 1.0,
float prob_Mutation = 0);
    ~AlgorithmeGenetique();

    virtual string nameAlgo() override { return "Genetique"; }

    virtual void nextSolution(Solution * next, Solution * current, Problem * prob)
override;

private:
    bool m_crossover;

    float m_prob_Mutation;

    int m_size_Of_Population;
    int m_size_Population_Child;

    Solution * m_best;
    int m_i_Worst;
    Solution * m_worst;
```

```
	vector<Solution *> population;

	Solution * getIndividuRandom(Problem * prob);

	void initialisationPopulation(Problem * prob);
	Solution * croisement(Problem * prob);

	void initialisationPopulationCO(Problem * prob);
	void croisementCO(Problem * prob, vector<Solution *> * population_Child);

	void mutation(Problem * prob, Solution * sol);

	Solution * getNewWorst();
	bool isPresent(Solution * sol);
	void newPopulation(Problem * prob, Solution * sol);

	bool isPresent(vector<int> * fixed, int val);
	void printPopulation();
};
```

AlgorithmeGenetique.cpp

```
#include "stdafx.h"
#include "AlgorithmeGenetique.h"

AlgorithmeGenetique::AlgorithmeGenetique(Problem * prob, bool co /*= true*/, float
size_Pop_Child /*= 1.0*/, float prob_Mutation /*= 0*/) :
	Algorithme(), m_prob_Mutation(prob_Mutation), m_crossover(co)
{
	m_best = nullptr;
	m_worst = nullptr;


	int d = ceil((double) prob->getNumberOfVertices() / prob->getNumberOfCenter());
	m_size_Of_Population = max(d * 4, (int) ceil(prob->getNumberOfVertices() / 3));
	m_size_Population_Child = (int) ceil(m_size_Of_Population * size_Pop_Child);


	if (m_crossover)
		initialisationPopulationCO(prob);
	else
		initialisationPopulation(prob);
}

void AlgorithmeGenetique::nextSolution(Solution * next, Solution * current, Problem *
prob)
{
	Solution * tmpSol;
	vector<Solution *> * population_Child = new vector<Solution *>;

	while (population_Child->size() < m_size_Population_Child)
	{
		if (m_crossover)
			croisementCO(prob, population_Child);
		else
		{
			tmpSol = croisement(prob);
			mutation(prob, tmpSol);
			population_Child->push_back(tmpSol);
		}
	}
```

```
	for each (Solution * var in (*population_Child))
	{
		newPopulation(prob, var);
	}

	next->copySolution(m_best);
}

void AlgorithmeGenetique::initialisationPopulation(Problem * prob)
{
	int const num_Vert = prob->getNumberOfVertices();
	vector<int> tmpSol = {0};

	int center = 0;
	int addinitialisation = 1;

	int i = 0, j = 1, k = 1;
	while (i < m_size_Of_Population)
	{
		if (k % prob->getNumberOfCenter() == 0)
		{
			population.push_back(new Solution(tmpSol, prob));
			if (m_best == nullptr || m_best->getFctObj() > population[i]-
>getFctObj())
				m_best = population[i];
			if (m_worst == nullptr || m_worst->getFctObj() < population[i]-
>getFctObj())
			{
				m_worst = population[i];
				m_i_Worst = i;
			}

			tmpSol.clear();
			i++;
		}

		if (j == num_Vert)
		{
			addinitialisation++;
			j = 0;
			center = -addinitialisation;
		}

		center = center + addinitialisation;

		if (center >= num_Vert)
		{
			center = center % num_Vert;
			if (addinitialisation % 2 == 0)
				center++;
		}

		tmpSol.push_back(center);

		j++;
		k++;
	}
}

Solution * AlgorithmeGenetique::croisement(Problem * prob)
{
```

```
	Solution * tmpSol = new Solution();
	Solution * tmpBest = new Solution();
	Solution * resetSol = new Solution();
	vector<int> * fixed;

	int s, r, getS;

	tmpSol->copySolution(getIndividuRandom(prob), prob);
	fixed = tmpSol->addSolutionReturnFixed(getIndividuRandom(prob)->getSolution());
	resetSol->copySolution(tmpSol);

	while (tmpSol->getSolution()->size() != prob->getNumberOfCenter())
	{
		s = (int) ceil(tmpSol->getSolution()->size() / 4.0);
		for (int i = 0; i < s; i++)
		{
			getS = tmpSol->getSolution()->size();
			while (isPresent(fixed, tmpSol->getValue(r = rand() % getS))) {}

			tmpSol->delValue(r);
			tmpSol->evaluateSolution(prob);
			if (tmpSol->getFctObj() < tmpBest->getFctObj())
				tmpBest->copySolution(tmpSol);

			tmpSol->copySolution(resetSol);
		}
		tmpSol->copySolution(tmpBest);
		resetSol->copySolution(tmpBest);
		tmpBest->resetFctObj();
	}

	delete tmpBest;
	delete resetSol;
	delete fixed;

	return tmpSol;
}

void AlgorithmeGenetique::initialisationPopulationCO(Problem * prob)
{
	for (int i = 0; i < m_size_Of_Population; i++)
	{
		population.push_back(new Solution(prob));
		if (m_best == nullptr || m_best->getFctObj() > population[i]-
>getFctObj())
			m_best = population[i];
		if (m_worst == nullptr || m_worst->getFctObj() < population[i]-
>getFctObj())
		{
			m_worst = population[i];
			m_i_Worst = i;
		}
	}
}

void AlgorithmeGenetique::croisementCO(Problem * prob, vector<Solution *> *
population_Child)
{
	Solution * p1 = new Solution();
	Solution * p2 = new Solution();

	p1->copySolution(getIndividuRandom(prob), prob);
```

```
        p2->copySolution(getIndividuRandom(prob), prob);

        int range = p1->getSolution()->size() - 1;
        int r = rand() % range + 1;

        vector<int> p1v1 = p1->getRangeValue(0, r - 1);
        vector<int> p1v2 = p1->getRangeValue(r, p1->getSolution()->size() - 1);
        vector<int> p2v1 = p2->getRangeValue(0, r - 1);
        vector<int> p2v2 = p2->getRangeValue(r, p2->getSolution()->size() - 1);

        Solution * c1 = new Solution(p1v1, p2v2, prob);
        Solution * c2 = new Solution(p2v1, p1v2, prob);

        population_Child->push_back(c1);
        population_Child->push_back(c2);
}
```