GOSSET Guillaume, LASOTA Quentin
PICHARD Thomas, SCION Gaël

Projet d'Optimisation Stochastique

Problème du voyageur de commerce

Document organique

Enseignant encadrant :

Mme KHEBBACHE Selma

Sommaire

I. Présentation du logiciel de résolution

Concernant notre choix de langage pour l'implémentation du recuit simulé, nous avons choisi le langage C++.

Notre logiciel de résolution contient 4 classes :

- Graph.hpp -> classe permettant de représenter la structure du graphe
- Node.hpp -> classe permettant de représenter un noeud du graphe
- Metaheuristique.hpp -> classe qui est chargée de manipuler le graph et de déterminer la solution du recuit simulé.
- Main.hpp -> classe qui est chargée de l'aspect interactif de l'application (chargement des données, saisie clavier,affichages/statistiques etc.)

Chaque classe contient différentes variables :

- Graph.hpp
  - Un vecteur de Node, permettant de stocker les noeuds
  - Une matrice permettant de stocker les distances moyennes nœud a nœud pour chaque nœud
  - Une chaîne de caractère pour stocker le nom du graph
  - Une chaîne de caractère pour stocker le type de graph
  - Une chaîne de caractère pour stocker le commentaire associé au graph
  - Un long permettant de stocker la dimension du graph
  - Deux chaînes de caractère, permettant de stocker le type de poids d'arc et le type de donnée
  - Un float pour stocker le facteur d'écart type
  - Deux type particuliers : std::random_device rd{} et std::mt19937 gen{rd()} utilisés pour la génération aléatoire selon la loi normale
- Node.hpp
  - Un indicatif unique [logicalNumber] sous forme numérique.
  - deux flottant simple précision permettant de stocker la position géographique du nœud dans l'espace.
  - Un booléen [alreadyTaken] permettant de stocker le fait que le nœud en question a déjà été pris dans le chemin sur lequel on travaille actuellement.
  - Un booléen permettant de stocker le fait que le sommet est le point de départ de l'algorithme (pour protéger son retrait lors des phases de perturbation de solution). (finalement non utilisé dans le code)
- Metaheuristic.hpp
  - Un vector de long permettant de stocker le chemin sur lequel on travaille couramment

- Un vector de long correspondant a la meilleure valeur que l'on a trouvé en général dans tout le recuit simulé
- Un vector de long correspondant a la meilleure solution qu'on a trouvé dans la phase d'intensification courante
- Une variable pour stocker la température actuelle du système
- Une variable pour stocker l'énergie du système.
- Deux floats le premier pour stocker l'énergie de la meilleur solution globale trouvée, le second pour stocker l'énergie de la meilleur solution de la phase d'intensification courante
- Une référence de Graph sur lequel on réalise les calculs
- Deux floats, le premier pour stocker la ligne base de température et le second pour stocker le facteur d'évolution de cette température
- Deux pointeurs de fonction qui stockent la fonction de génération initiale et la fonction d'exploration du voisinage a utiliser sur l'instance
- Une chaîne de caractère pour stocker le chemin du fichier d'exportation des résultats (au format csv)
- Plusieurs vecteurs de float permettant de stocker l'évolution de la température, de l'énergie de solution courante, de l'énergie de la meilleur solution globale trouvée et de l'énergie de référence pour la phase d'intensification courante
- Un bool permettant de stocker si l'on réalise ou non une exportation des résultats à la volé
- Une variable de type std::ofstream pour garder un flux de sortie vers le chemin stockant les résultats dans le cas d'une écriture à la volé

Chaque classe contient différentes méthodes :

- Graph.hpp
  - une méthode pour récupérer la référence d'un nœud du graph selon son ID (opérateur []).
  - une méthode pour récupérer le nœud non pris le plus proche géographiquement.
  - une méthode pour récupérer la distance entre 2 à n nœuds.
  - Une méthode getPathWeigth() qui permet de récupérer la longueur d'un chemin donnees en tenant compte de l'aléatoire
  - Une méthode getEffectiveDistance() qui étant donnée deux noeuds permet de récupérer la distance entre ces deux noeuds en incluant la composante aléatoire
  - Une méthode SetTaken permet d'indiquer qu'un noeud est déjà pris dans la solution courante

    - Une méthode ReinitTaken permet de réinitialiser l'ensemble des nœuds d'un graph et de les remettre a l'état non pris.
- Node.hpp
    - Une méthode pour mettre à jour ce statut.
    - Une méthode pour calculer la distance par rapport à un autre nœud.
- Metaheuristic.hpp
    - Une méthode globale [solv()] permettant d'appliquer le recuit simulé sur le Graph.
    - Une méthode d'exportation des résultats, vers le chemin spécifié en entrée
- Operators.hpp (Bibliothèque)
    - Stock les différentes fonctions d'initialisations de la solution initial et de randomisation pour l'exploration du voisinage
        - Initialisation gloutonne
        - Initialisation Random
        - Exploration avec un échange d'un couple de valeur unique
        - Exploration avec des échanges multiples, dont le nombre d'échanges dépend d'une probabilité décroissante
        - Exploration avec un échange d'un bloc de trois valeurs avec un autre bloc de trois valeurs prise aléatoire ailleurs.

Notre application comprend deux mains :
- main.cpp
- mainFinale.cpp

Le premier main est un main classique en c++, ne prenant aucun paramètre à l'exécution, mais vous pouvez modifier les paramètres d'exécution à l'intérieur de ce fichier avant compilation de l'application.
Le second main prend l'ensemble des paramètres de lancement dans la ligne d'exécution.

Pour compiler le projet voici les deux commandes pour compiler avec le premier et le second main :

```
g++ -std=c++17 -Wall -Werror Graph.cpp Metaheuristic.cpp Node.cpp Operators.cpp
main.cpp -o recuit
```

```
g++ -std=c++17 -Wall -Werror Graph.cpp Metaheuristic.cpp Node.cpp Operators.cpp
mainFinal.cpp -o recuit
```

Pour l'exécution du second main, la commande de lancement est décrite ci-dessous :

```
./a.out 200 0.5 0.99999 0.025 ../data/tsplib_dataset/berlin52.tsp ./basic_glouton 1 0 0
```

Les différents paramètres sont dans l'ordre :

- Température de départ
- Température d'arrivée
- Facteur de diminution de la température
- Facteur d'étalement autour de la moyenne (exprimé sous la forme d'un multiplicateur appliqué à la moyenne)
- Chemin vers le fichier source
- Nom du fichier d'export des résultats
- Choix de la méthode de la génération de solution initial (0 -> Aléatoire, 1 -> Glouton)
- Méthode de randomisation pour l'exploration du voisinage (0 -> Échange simple, 1 -> Echange multiple, 2 -> Echange par bloc de trois)
- 0-> Export des résultats a la fin du calcul, 1-> Export à la volé au fur et à mesure du calcul

Pour accélérer l'exécution de la partie stochastique du recuit simulé, nous avons utilisé la librairie openmpi pour paralléliser une partie du calcul. Pour compiler le projet et activer cette partie il faut utiliser la commande :

```
g++ -std=c++17 -fopenmp -Wall -Werror Graph.cpp Metaheuristic.cpp Node.cpp Operators.cpp
main.cpp -o recuit
```

Pour l'exécution il faut juste indiquer le nombre de threads que vous souhaitez créer (exemple avec 4 threads):

```
OMP_NUM_THEADS=4 ./a.out 200 0.5 0.99999 0.025 ../data/tsplib_dataset/berlin52.tsp
./basic_glouton 1 0 0
```

Générateur de sous graphes pour CPlex :

Dans le dossier tools, nous avons deux programmes utiles pour la génération des graphes sous CPlex. Le premier est dataset_gen.py, qui s'occupe de générer des graphes sous un des formats de données récupérés pour le TSP (avec les coordonnées x,y des points). Il suffit de l'exécuter avec en paramètre le nombre de sommets du graphe :

```
python dataset_gen.py 6 nomGraph
```

Pour le deuxième programme, nommé subgraph_gen.py, il permet de générer le format attendu par CPlex (OPL), et optionnellement de générer le fichier des subtours (sous-ensembles possibles). Il faut l'exécuter avec la commande :

```
python subgraph_gen.py nomGraph.tsp {0,1}
```

Le dernier paramètre (mit à 0 ou 1) indique si l'on veut générer les sous-ensembles.

Générateur de graphiques pour C++ :

Nous avons créé un générateur de graphiques en python pour représenter les résultats, dans le dossier "tools" , nommé generateGraph.py.
La commande de lancement est `python generateGraph.py`
Le générateur va automatiquement traiter les 6 fichiers CSV se trouvant dans le dossier recuitSimule et va générer les graphiques (en png) dans le même dossier que le fichier python
(Le générateur demande d'avoir installé au préalable, les librairies Pandas, Matplotlib et numpy sur votre machine)

II. Présentation et analyse des résultats CPlex

Pour lancer les résultats pour différents graphes, il suffit de lancer le modèle main.mod, qui invoque, pour les données voulues, les modèles déterministes et stochastiques. Les résultats sont ensuite inscrits dans un fichier resultsMain.csv.

Pour lancer la simulation (qui fait varier alpha et la variance), il suffit de lancer le modèle simu.mod qui va invoquer les modèles déterministes et stochastiques pour chaque couple <alpha,variance>. Les résultats sont écrits dans resultsSimu.csv.

Pour modifier les instances testées, il faut respectivement modifier dataMain.dat (il est possible de générer des graphes avec n sommets grâce aux outils du dossier tools) et dataSimu.dat (où l'on indique les instances de alpha et celles de la variance, où les données sont un pourcentage de la moyenne pour l'écart-type).

A - Résultats de main.mod

(optimum | temps d'exécution en ms)

Graphe	Déterministe		Stochastique		Déterministe MTZ		Stochastique MTZ	
d6	564.567	17	564.567	21	564.567	104	564.567	28

d10	541.330	53	541.330	63	541.330	61	541.330	43
d20	728.686	222145	728.686	227527	728.644	346	728.681	217

Les noms des graphes comprennent leur nombre de sommets.
On remarque d'abord qu'on retrouve le même optimum (ou pratiquement le même avec d20) avec le modèle déterministe et stochastique, et peu importe la modélisation (subtour et MTZ).

On remarque que la modélisation avec sous-ensemble est efficace pour les toutes petites instances, mais très très lente pour les instances moyennes (et grandes, par conjecture). En effet, même si la donnée de temps d'exécution indique le contraire, nous avons passé environ 8 heures pour obtenir les résultats, principalement pour l'exécution de d20 avec le modèle subtours (donc environ 4 heures pour déterministe et 4 heures pour stochastique).

La modélisation MTZ (avec rang), quant à elle, est très **robuste** à l'augmentation du nombre de sommets, du moins pour un nombre de sommets modéré.
Il est donc intéressant de lancer le modèle MTZ avec des graphes contenant plus de sommets.

B - Résultats de main.mod (MTZ, beaucoup de sommets)

Graphe	Déterministe MTZ (optimum)	Temps d'exécution en ms	Stochastique MTZ (optimum)	Temps d'exécution en ms
d6	564.567395	28	564.567395	31
d10	541.3303	53	541.3303	40
d20	728.644053	314	728.681256	205
d50	1113.88197	27340	1113.88796	11532
d100	1478.95084	240105	1476.57092	240057

Avec ces résultats nous pouvons remarquer que le modèle déterministe et stochastique obtiennent des valeurs semblables (voire identiques pour d6 et d10). En ce qui concerne le temps d'exécution, il reste relativement raisonnable (du moins en comparaison avec la méthode des subtours), et l'exécution pour le graphe d100 arrive à la limite de temps que nous avons fixée.

Ensuite, on a observé que le résultat déterministe est semblable au résultat stochastique. Cela peut être dû à l'unicité du graphe que l'on génère pour chaque nombre de sommets. On tente alors de générer plusieurs graphes pour un même nombre de sommets (assez petit), et observer si on retrouve une résultat déterministe et stochastique égal.

C - Résultats sur plusieurs générations pour un même nombre de sommets

Graphe	Stochastique	Stochastique MTZ		Graphe	Stochastique	Stochastique MTZ
d6	564.567395	564.567395		d10	541.3303	541.3303
d6_1	541.365386	541.365386		d10_1	503.061597	503.061597
d6_2	557.204433	557.204433		d10_2	522.062307	522.062307
d6_3	398.007499	398.007499		d10_3	578.643648	578.643648
d6_4	412.680827	412.680827		d10_4	507.280046	507.280046
d6_5	362.758488	362.758488		d10_5	572.715152	572.715152
d6_6	412.737721	412.737721		d10_6	687.927917	687.927917
d6_7	390.809555	390.809555		d10_7	622.002342	622.002342
d6_8	478.697116	478.697116		d10_8	556.137277	556.137277
d6_9	459.437524	459.437524		d10_9	440.569529	440.569529
d6_MTZ	564.567395	564.567395		d10_MTZ	541.3303	541.3303
d6_1_MTZ	541.365386	541.365386		d10_1_MTZ	503.061597	503.061597
d6_2_MTZ	557.204433	557.204433		d10_2_MTZ	522.062307	522.062307
d6_3_MTZ	398.007499	398.007499		d10_3_MTZ	578.643648	578.643648
d6_4_MTZ	412.680827	412.680827		d10_4_MTZ	507.280046	507.280046
d6_5_MTZ	362.758488	362.758488		d10_5_MTZ	572.715152	572.715152
d6_6_MTZ	412.737721	412.737721		d10_6_MTZ	687.927917	687.927917
d6_7_MTZ	390.809555	390.809555		d10_7_MTZ	622.002342	622.002342
d6_8_MTZ	478.697116	478.697116		d10_8_MTZ	556.137277	556.137277
d6_9_MTZ	459.437524	459.437524		d10_9_MTZ	440.569529	440.569529

On observe donc que pour un graphe donné, les quatres optimums sont bien égaux, peu importe la génération du graphe.

D - Résultats de simu.mod

On lance la simulation sur le graphe d10 (contenant 10 sommets). On fait varier les valeurs de alpha et de stvDeviaCoef (coefficient qui multiplie la moyenne pour donner l'écart-type). On trouve le même optimum pour tous les couples <alpha, stdDeviaCoef>, soit 541.3303. Les temps d'exécutions sont présentées ci-dessous :

Temps d'exécution (ms) Stochastique subtour

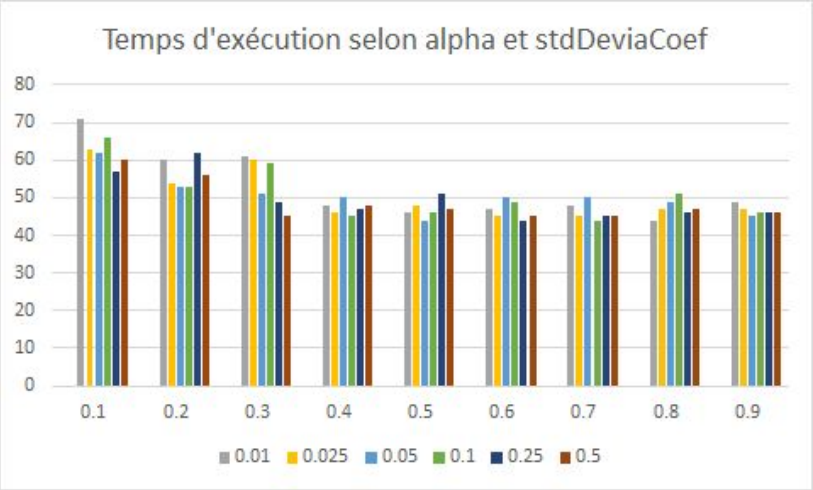
alpha stdDeviaCoef	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.01	71	60	61	48	46	47	48	44	49
0.025	63	54	60	46	48	45	45	47	47
0.05	62	53	51	50	44	50	50	49	45
0.1	66	53	59	45	46	49	44	51	46
0.25	57	62	49	47	51	44	45	46	46
0.5	60	56	45	48	47	45	45	47	46

Temps d'exécution (ms) Stochastique MTZ

alpha stdDeviaCoef	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.01	29	40	32	35	33	34	25	32	33
0.025	40	33	34	33	31	36	30	28	29
0.05	28	30	32	32	37	27	28	30	31
0.1	38	33	45	36	29	27	25	25	30
0.25	29	32	64	32	26	31	37	26	27
0.5	33	50	38	27	36	28	29	26	30

On observe qu'il n'y a pas d'impact pour la solution optimale.

Néanmoins, on peut observer une différence au niveau du temps d'exécution, même si elle peut sembler négligeable. On les représentent sur un graphique :



On observe que la valeur de stdDeviaCoef (différentes valeurs de couleur) ne semble pas avoir de répercussion systématique sur le temps d'exécution. Néanmoins, on peut observer que plus alpha est petit, plus le temps d'exécution semble long (les barres pour alpha=0.1 à 0.3 sont plus hautes, il semblerait donc y avoir une relation).
Par contre, les couples sont exécutés dans l'ordre des alpha croissants (puis, pour une valeur d'alpha, selon les stdDeviaCoef croissant), donc cela pourrait être un ralentissement dû au début de l'exécution.

On conclut qu'il n'y a pas de relation concluante entre alpha ou stdDeviaCoef et l'optimum ou le temps d'exécution. Cela peut être dû au graphe utilisé, qui est très petit, mais le fait est que le modèle à sous-ensemble explose rapidement.

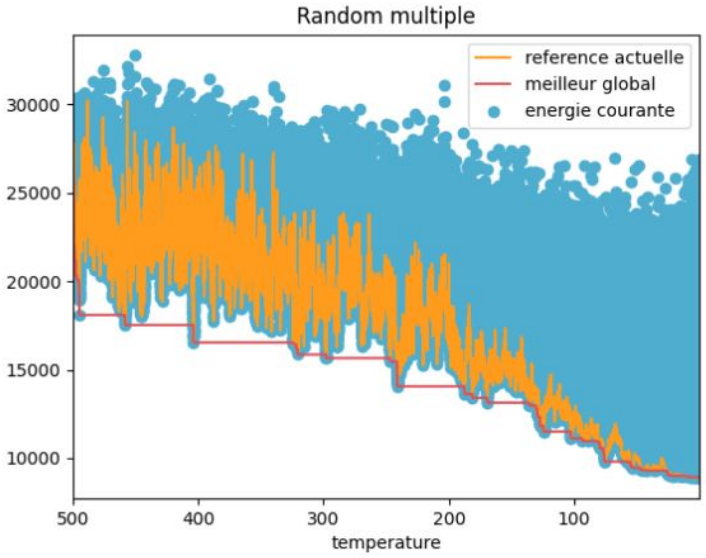
III. Présentation et analyse des résultats C++

A. Présentation des résultats

Nous avons exécuté notre recuit simulé sur un certain nombre de jeux de test, pour pouvoir comparer les résultats obtenus avec les résultats obtenus par CPLEX et déterminer si notre implémentation était correcte. Les jeux de test utilisés sont :

- d6
- d10
- d20
- d50
- d100
- ali535
- ulysses22
- d198
- d198 (avec des paramètres d'analyse particulier)

Un premier exemple ci-dessous avec un graphique obtenu grâce aux résultats du recuit simulé sur le fichier de données berlin52, pour détailler les différents éléments que vous pouvez retrouver sur tous les graphiques qui suivront :



Le titre peut être décomposé en deux parties (dans cet exemple “random” et “multiple”). En première partie, il s’agit de la méthode de génération de la solution initiale (dans ce cas une méthode aléatoire). Pour la deuxième partie, il s’agit de la méthode de randomisation pour l’exploration du voisinage (dans ce cas un échange multiple).

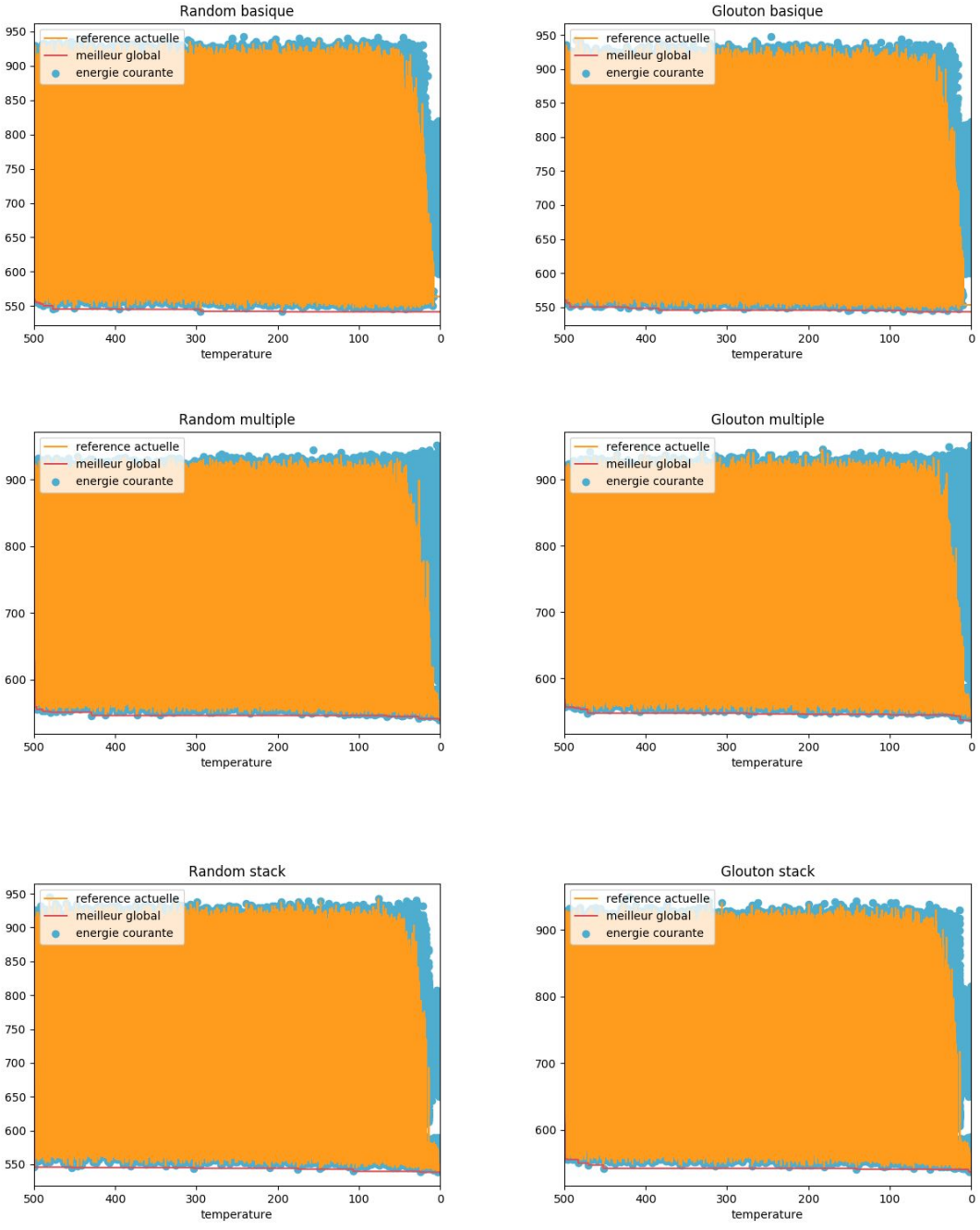
A l’intérieur du graphique, vous pouvez voir l’évolution de la valeur de la référence actuelle en orange, l’évolution du meilleur global en rouge et les valeurs de l'énergie courante en nuage de points en bleu. Dans ce cas, nous pouvons bien voir que le meilleur global diminue de manière cohérente en suivant la diminution de la température pour atteindre la solution globale retournée par le programme C++.

Concernant les paramètres d'exécution par défaut, nous avons une température de départ à 500, une température minimum d’arrivée à 0.05, un facteur de diminution de la température de 0.99999 et un facteur d’étalement autour de la moyenne de 0.025.

Ci-dessous, vous pourrez voir des graphiques représentant l’évolution du calcul du recuit simulé sur l’ensemble des fichiers définis précédemment :

(l’axe des y indique la distance du chemin obtenu)

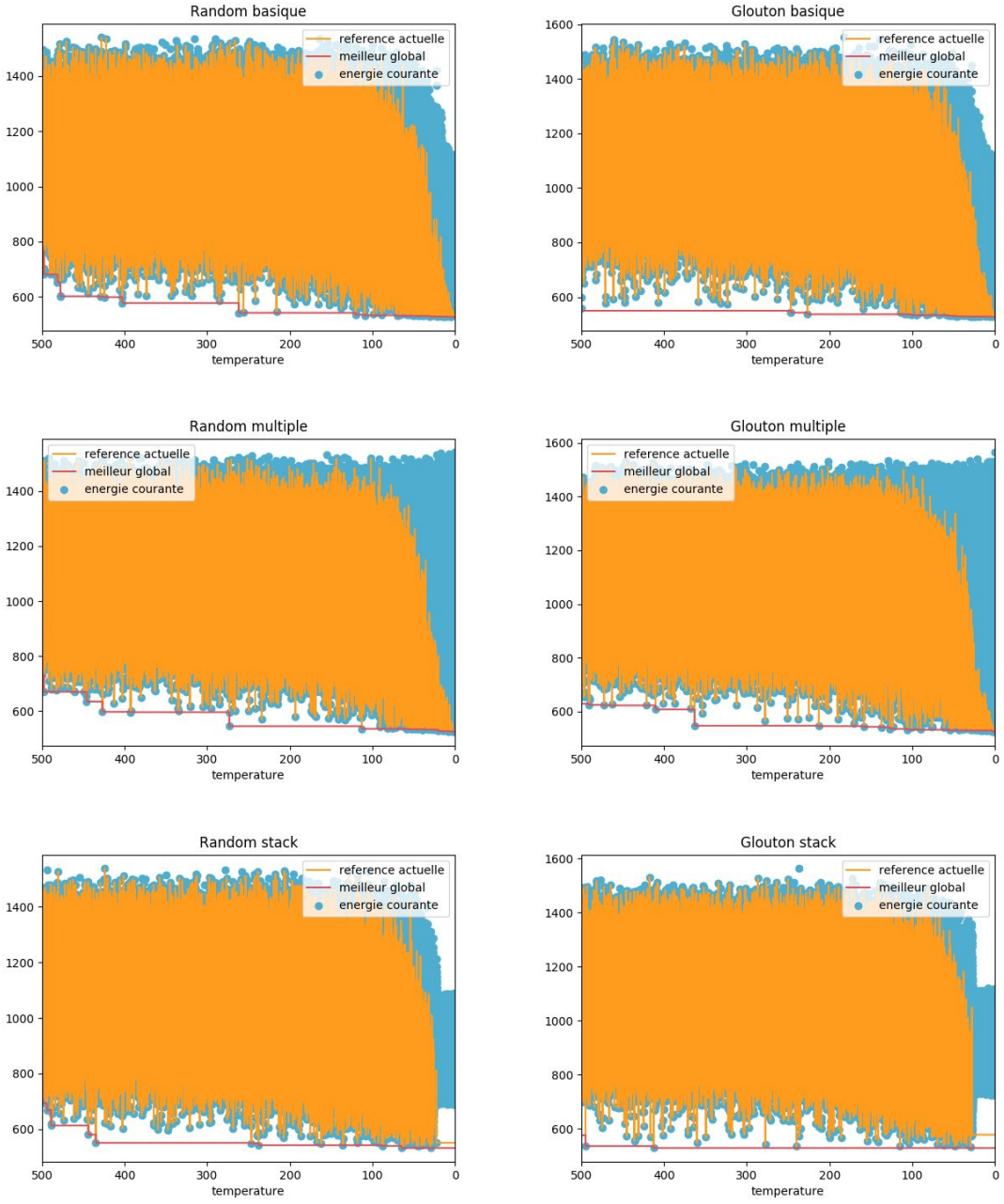
a. Evolution du calcul avec d6



Meilleur résultat : 564.567 (random basique)
Résultat sur CPlex : 564.567395

Efficacité : optimum trouvé, pour un temps d'exécution de 2.84s (exports inclus).

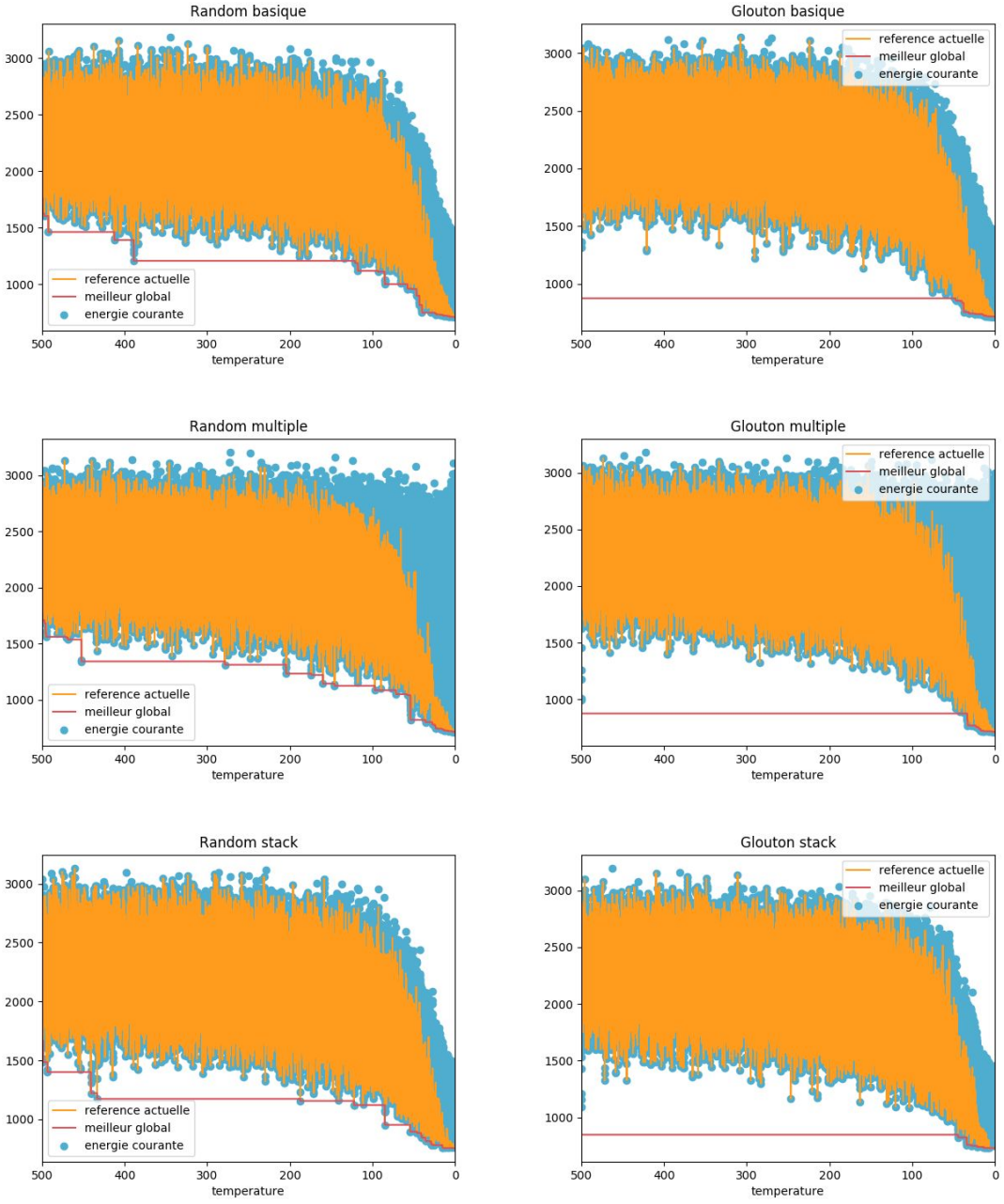
b. Evolution du calcul avec d10



Meilleur résultat : 541.33 (random basique)
Résultat sur CPlex : 541.3303

Efficacité : optimum trouvé, pour un temps d'exécution de 3.35s (exports inclus).

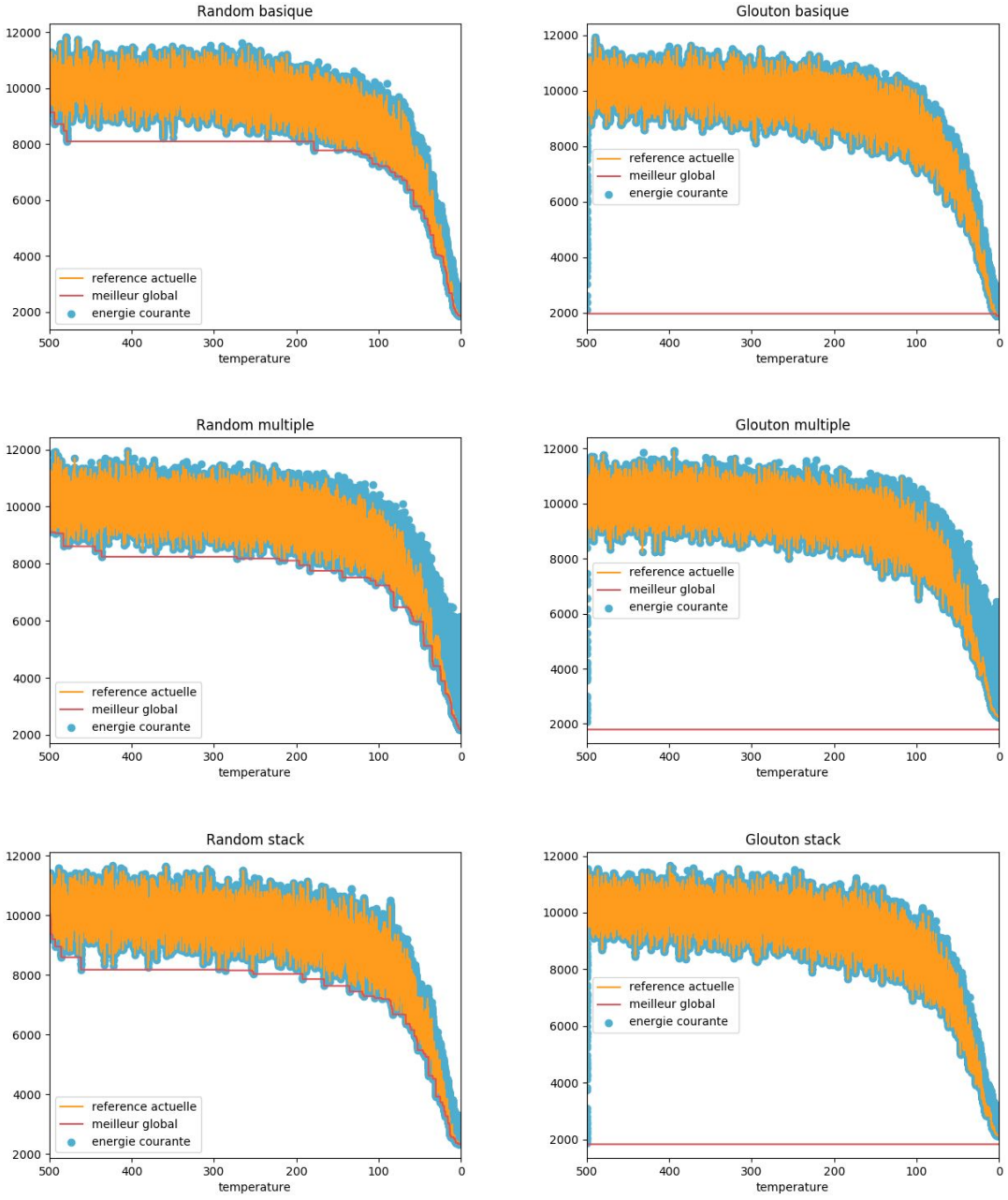
c. Evolution du calcul avec d20



Meilleur résultat : 728.686 (random basique)
Résultat sur CPlex : 728.644053

Efficacité : environ 100% de l'optimum, pour un temps d'exécution de 4.76s (exports inclus).

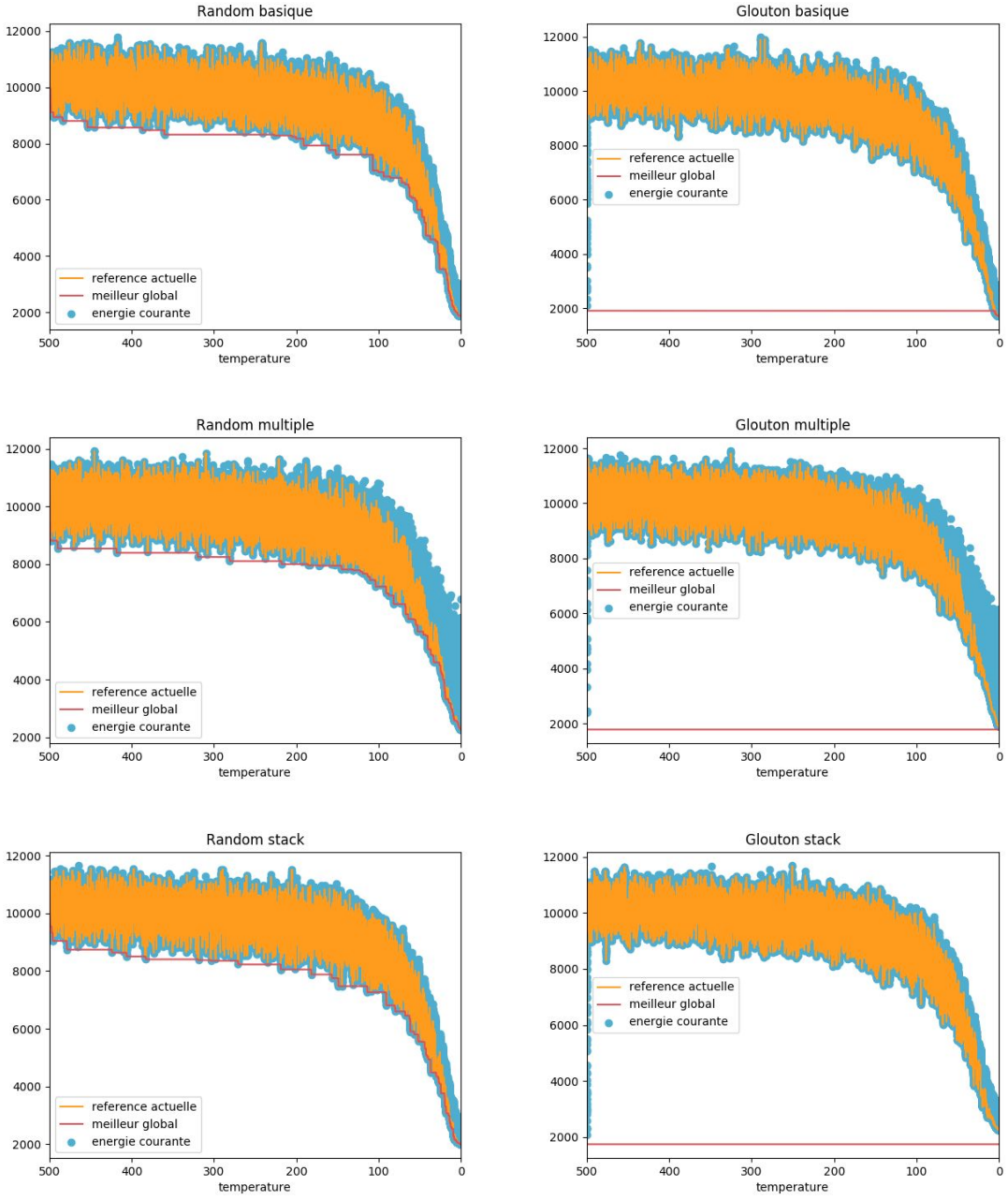
d. Evolution avec d50



Meilleur résultat : autour de 1216 (random basique)
Résultat sur CPlex : 1113.88197

Efficacité : environ 91.6% de l'optimum, pour un temps d'exécution d'environ 9s (exports inclus).

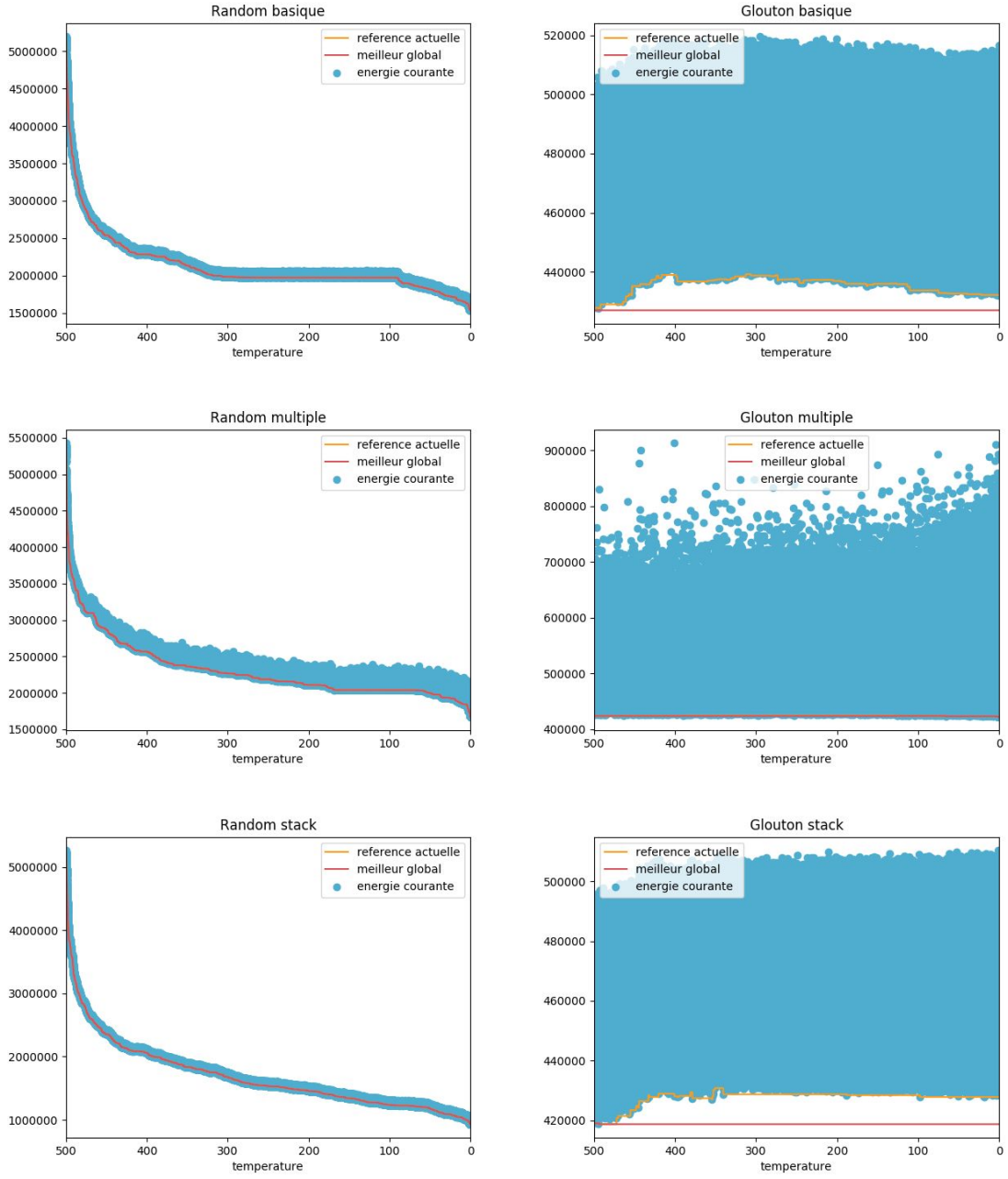
e. Evolution avec d100



Meilleur résultat : autour de 1800 (glouton basique)
Résultat sur CPlex : 1478.95084

Efficacité : environ 82% de l'optimum, pour un temps d'exécution d'environ 17s (exports inclus).

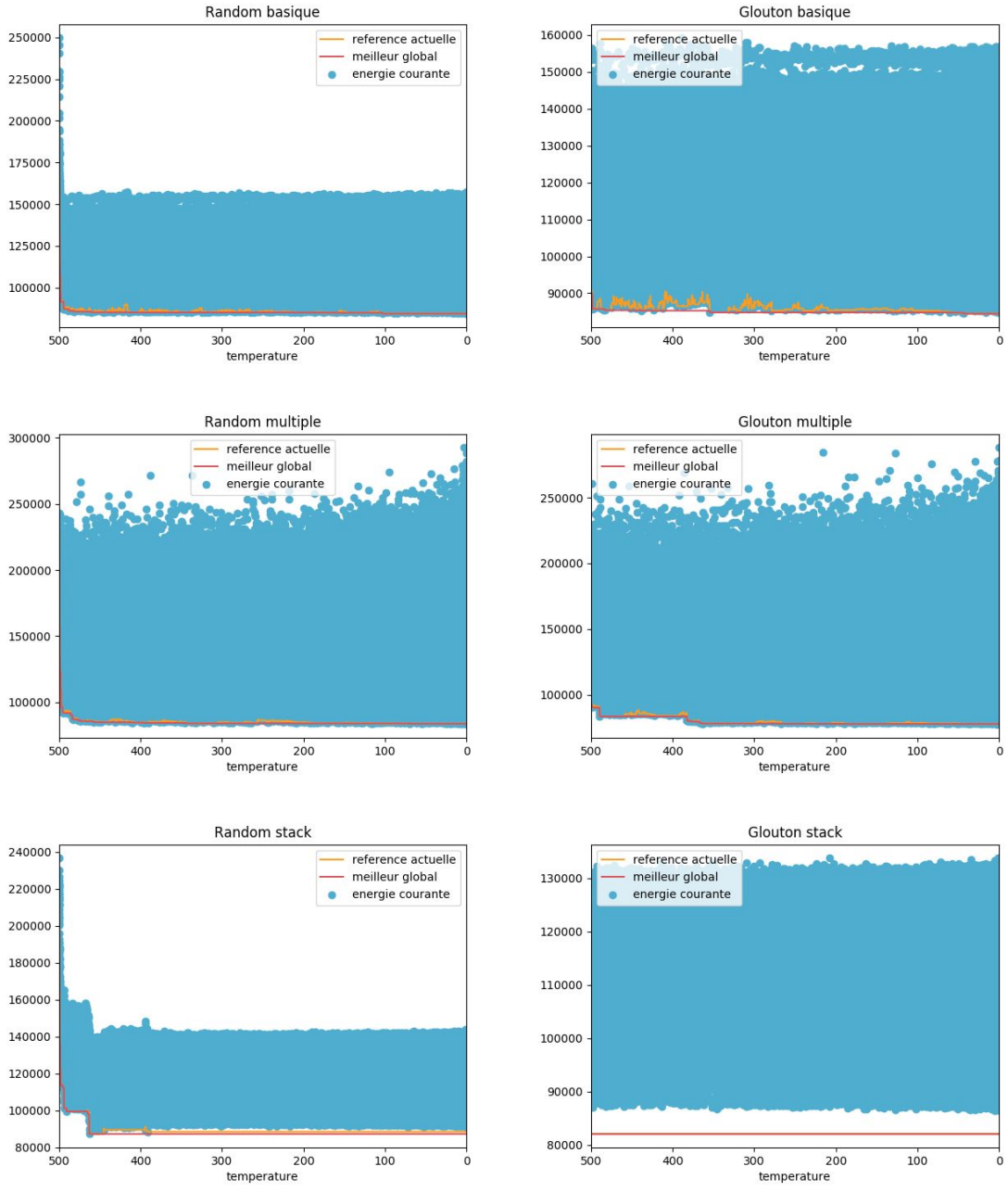
f. Evolution avec ali535



Meilleur résultat : 433000 (glouton basique)
Résultat de la littérature : 202339

Efficacité : environ 46.7% de l'optimum, pour un temps d'exécution de 77s (exports inclus).

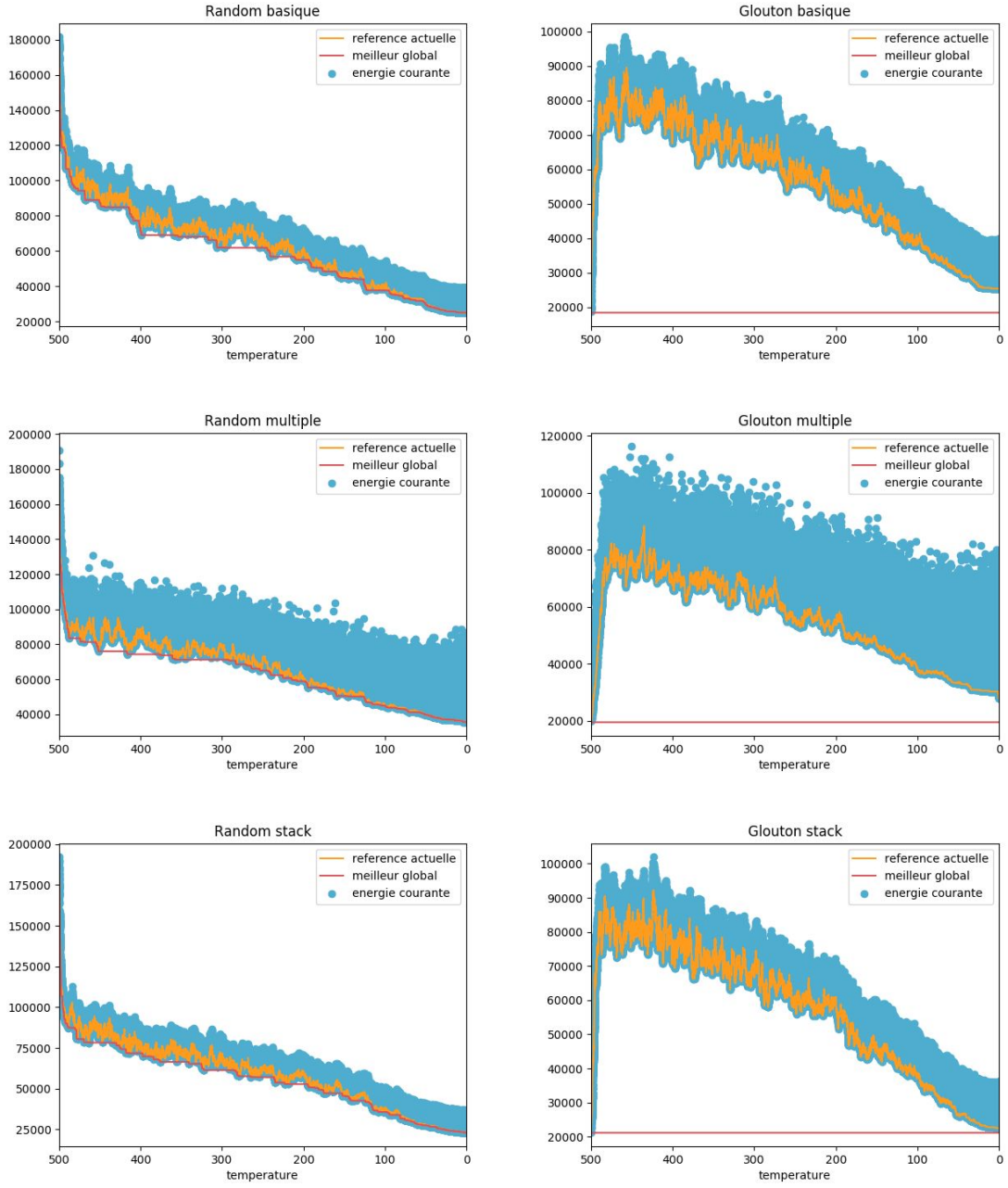
g. Evolution sur ulysses22



Meilleur résultat : environ 79000 (glouton multiple)
Résultat de la littérature : 7013
(mais l'écart est dû à une différence de calcul pour les coordonnées géographiques)

Efficacité : environ 8.9% de l'optimum, pour un temps d'exécution de 5.5s (exports inclus).

h. Evolution sur d198

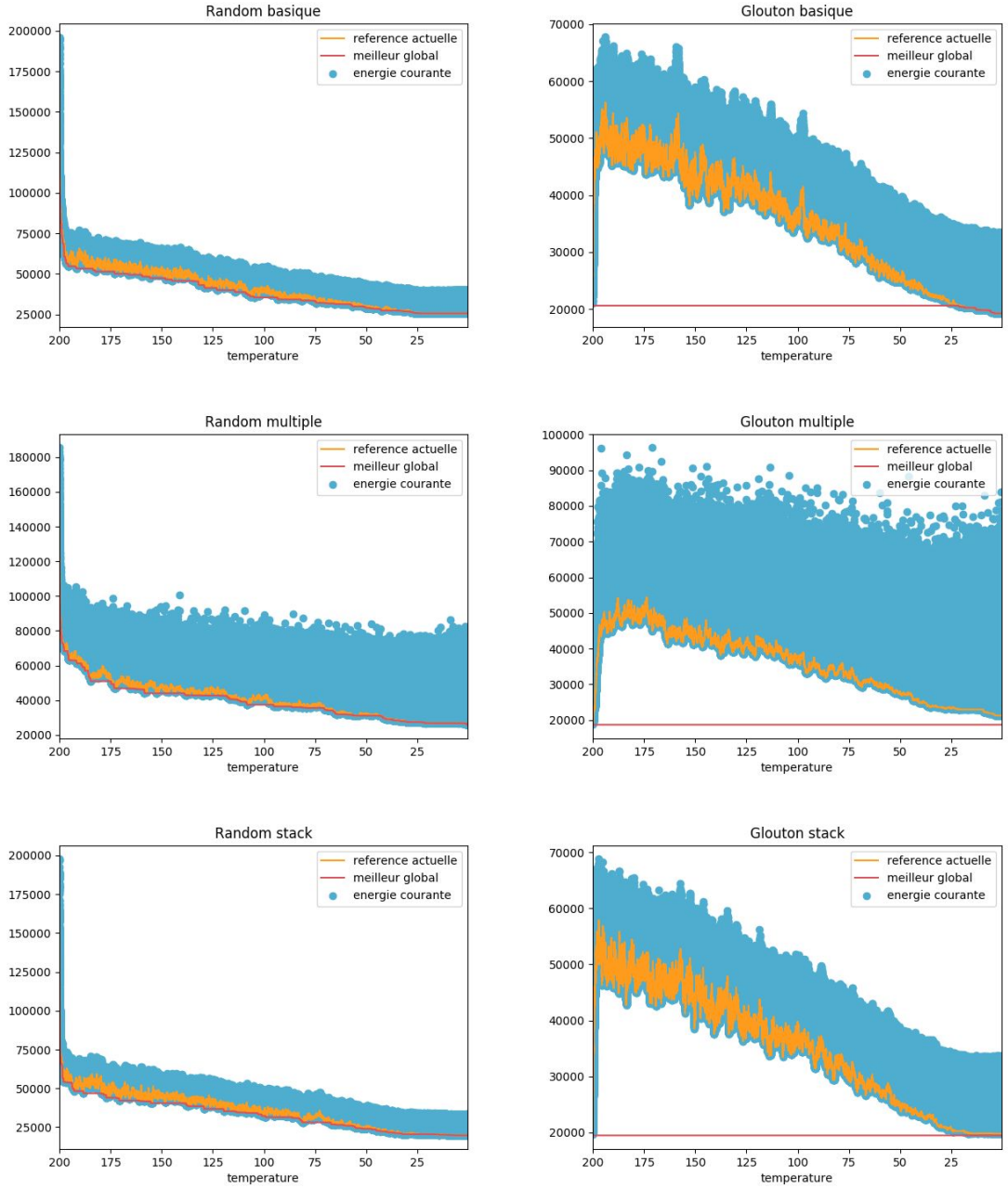


Meilleur résultat : environ 20000 (glouton stack)
Résultat de la littérature : 15780

Efficacité : environ 79% de l'optimum, pour un temps d'exécution de 28.85s (exports inclus).

i. Evolution sur d198 (paramètres optimisés)

(avec une température initiale de 200, une température finale de 0.5 et un facteur de réduction la température de 0.999999)



Meilleur résultat : un peu plus de 19000 (glouton stack)
Résultat de la littérature : 15780

Efficacité : environ 83% de l'optimum, pour un temps d'exécution de 220s à 230s.

B. Analyse

Tout d'abord, on peut remarquer que dans les solutions que nous avons obtenues, le fonctionnement de l'algorithme ressemble bien à ce qu'on pourrait attendre. La solution courante oscille énormément au sein de l'espace des solutions. Quant à elle, la solution de référence courante oscille de manière plus ou moins prononcée dans l'espace de solution en fonction de si la solution s'améliore ou si une solution moins bonne a été acceptée, et elle tend à se stabiliser progressivement au fur et à mesure de la réduction de température. Enfin la meilleure solution globale est bien strictement décroissante au fil des découvertes de nouvelles meilleures solutions.

Avec ces différentes exécutions, on peut principalement remarquer que les fonctions de randomisation permettent d'explorer une bonne partie de l'espace des solutions (visible grâce à la zone bleue, relativement uniforme, qui apparaît dans la plupart des graphes, indiquant qu'on explore des solutions très diverses avec des énergies associées très variées).

En revanche, il faudrait sans doute les améliorer pour obtenir de meilleurs résultats, tout particulièrement pour les algorithmes gloutons. En effet, dans plusieurs cas de tests que nous avons mis dans les graphiques ci-dessus, on constate que l'heuristique gloutonne parvient à trouver un très bon résultat au départ mais la randomisation telle qu'elle est faite dégrade très rapidement cette solution et met beaucoup de temps à ré-atteindre des résultats au moins équivalent par la suite. Dépendamment des cas, l'algorithme y parvient et on arrive à avoir une légère amélioration supplémentaire. Cependant, dans plusieurs autres cas, on restera sur la solution du glouton initial.

On remarque que sur les petits graphes des exemples (qui ont été utilisés pour comparer avec les données obtenues du CPlex), on a de très fortes variations du résultat de référence courant, et cela peut s'expliquer par le fait que les distances entre les noeuds sont relativement courtes dans le cadre de ces exemples, ce qui se traduit par une variation d'énergie relativement faible pour les permutations, augmentant leurs chances d'acceptation. La réduction très rapide des variations de la courbe en fin d'exécution corrobore cette hypothèse.

On peut également constater que prendre une solution initiale aléatoire peut, en fonction des cas, donner une pénalité de départ beaucoup plus grande (jusqu'à un facteur 10 par rapport à l'optimal). Par conséquent, la solution finale que l'on obtient en temps raisonnable (de l'ordre de 5 minutes) est en général moins bonne que ce qu'on arrive à obtenir en faisant un départ à partir du glouton (car la plupart de la haute température est "gaspillée" pour récupérer la pénalité de base de la solution).

On peut aussi remarquer que la multiplicité des opérateurs de randomisation est pertinente, car en fonction des topologies de graphe sur lesquelles on exécute le système, certains opérateurs permettent d'avoir une meilleure convergence ou une convergence plus rapide qu'avec d'autres. De même, partir d'une base différente entre aléatoire ou glouton (du moins pour ceux que l'on a développé) peut présenter des avantages ou des désavantages en fonction des topologies de graphe.

On constate cependant que les résultats obtenus sont en général bien en dessous des valeurs optimales, avec un écart qui tend à se creuser au fur et à mesure de l'augmentation de la taille du graphe considéré, de part la plus grande taille de l'espace des solutions à explorer par voisinage. Pour améliorer encore plus les résultats dans ce cas, il serait nécessaire de créer des opérateurs d'exploration encore plus efficaces pour permettre de mieux diriger l'évolution.

Une autre perspective d'amélioration pourrait être de retravailler les valeurs de paramétrage de la métaheuristique pour tenter d'en améliorer les résultats. Cependant, en essayant d'allonger un peu le temps de l'algorithme en réduisant la réduction de température, les gains obtenus restaient globalement assez réduits, le gain de performance par ce seul biais pourrait se traduire par une forte hausse du temps de calcul nécessaire.

En revanche, sur les plus petits graphes utilisés dans CPlex, nous avons réussi à obtenir des résultats au final très proches de l'optimal que nous avions obtenu en parallèle.

On peut enfin remarquer un phénomène assez intéressant dans le cas du mélange par bloc sur d10 (tout petit graphe) : dans l'exécution effectuée, on remarque qu'en dessous d'environ 20 degrés, l'espace des solutions parcouru se restreint subitement. Cela peut être dû à une situation particulière qui piège la solution dans un mauvais minimum local dont il sera plus dur de sortir, étant donné que l'on déplace à chaque fois deux blocs de 3 éléments disjoints sur un total de 10. Cet opérateur de randomisation n'est donc pas très adapté à des graphes de petite taille.

De la même manière, il peut enfin être intéressant de remarquer que l'on aura une progression de la solution très douce dans les graphes de grande taille comme d198 ou ali535, là où on aura une progression beaucoup plus incrémentale (par palier successifs assez brutaux) dans les graphs de petite taille comme ulysses22 ou berlin52.

III. Conclusion

En conclusion, au travers des résultats obtenus avec CPlex et notre programme C++, nous pouvons dire que le calcul de la solution exacte avec CPlex est très précise de part la nature du calcul, mais il s'agit d'un calcul très lourd et très lent (8h30 de calcul pour d20 par exemple, avec la méthode des sous-ensembles).
Par contre, l'implémentation du recuit simulé avec C++ est tout à fait adaptée pour calculer la solution dans un cas où il y a un grand nombre de points, car nous obtenons des résultats convenables malgré une grande quantité de points pour une fraction de temps bien acceptable par rapport à CPlex (même en comparant avec le modèle Miller-Tucker-Zemlin).
Cependant, pour obtenir des performances vraiment acceptables avec le recuit simulé sur des graphes à plus grande échelle (200 sommets ou plus), il est nécessaire de faire une phase d'optimisation fine des hyper-paramètres et de trouver des opérateurs de perturbation permettant de parcourir l'espace des solutions de façon très efficace.