TP 1 – Métaheuristique

Bellanger Clément

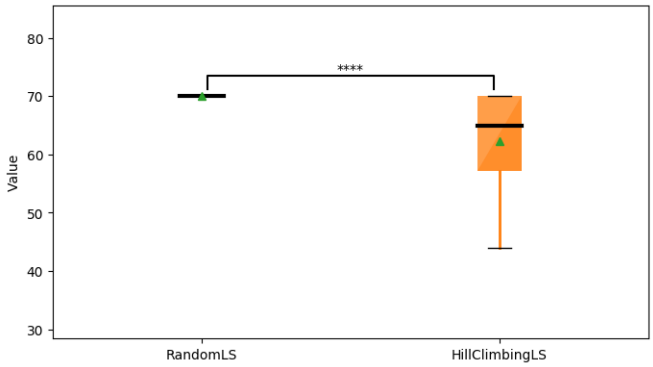
Résultats

Les paramètres utilisés sont 3000 pour le nombre d’itérations et d’évaluations et 30 exécutions (les paramètres par défaut)

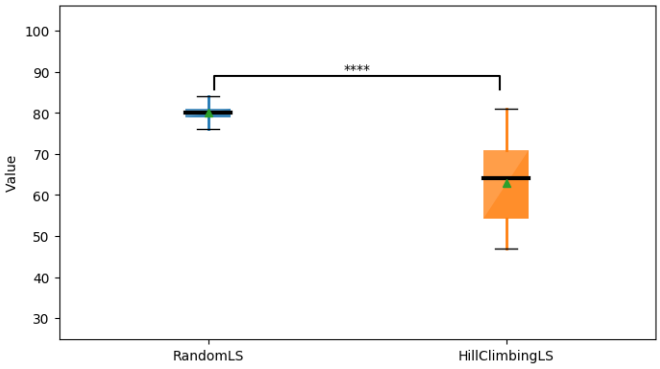
Les problèmes ont été testés dans toutes les tailles.

Problème – Sac à Dos

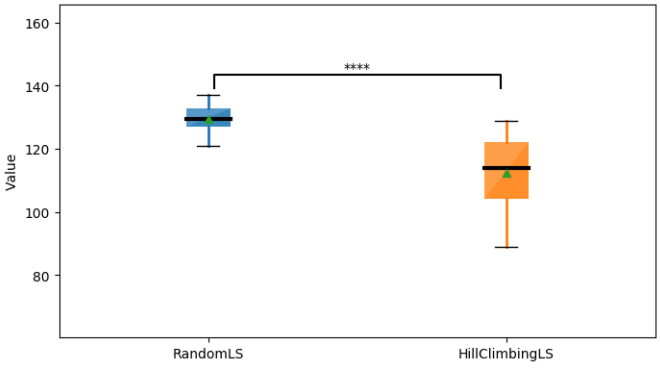
Taille – small



Taille – medium

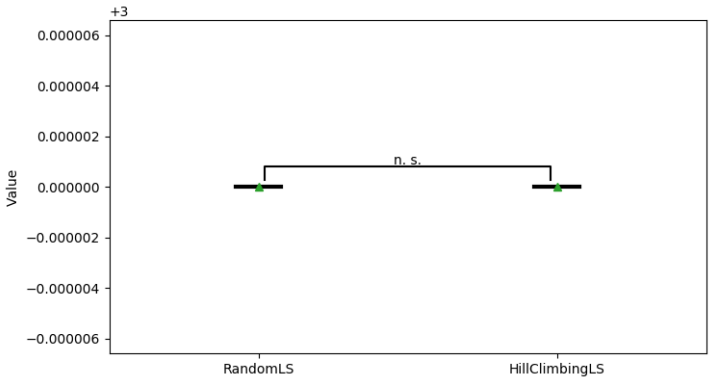


Taille - large

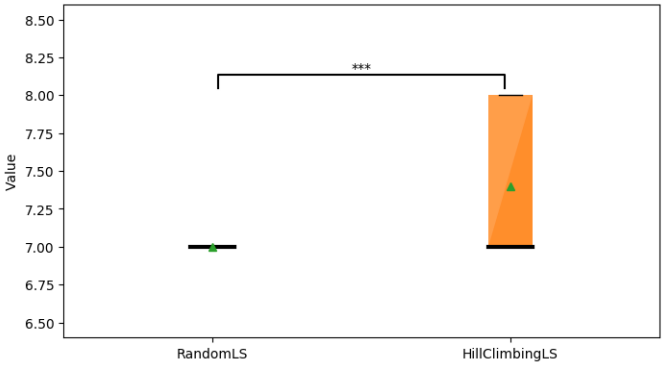


Problème – Set Covering

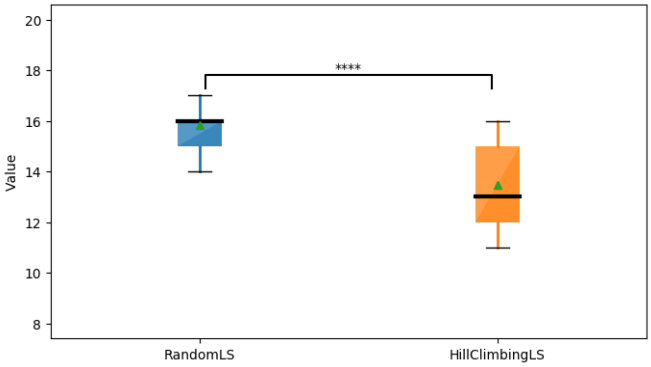
Taille – small



Taille – medium

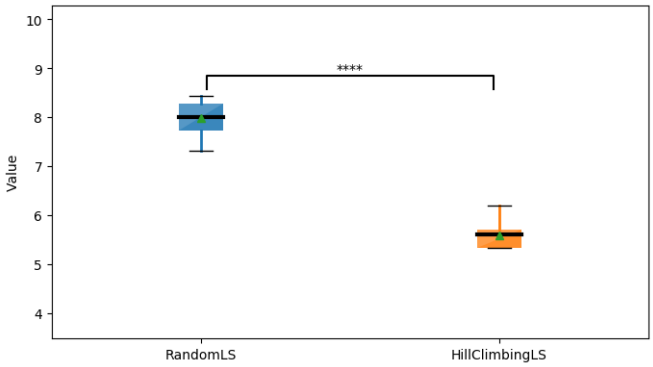


Taille - large

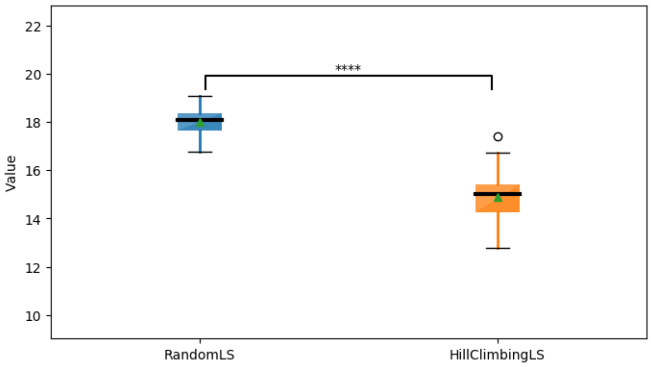


Problème – Voyageur de commerce (TSP)

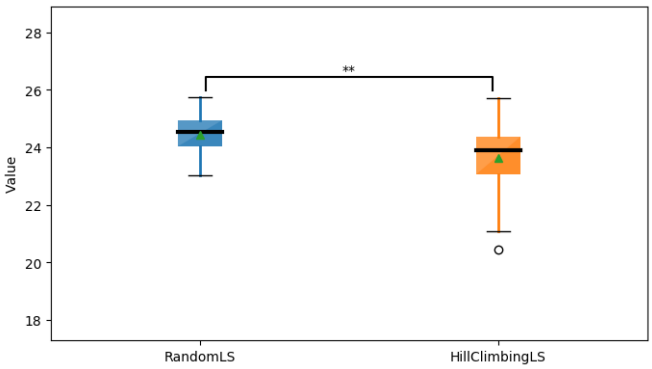
Taille – small



Taille – medium

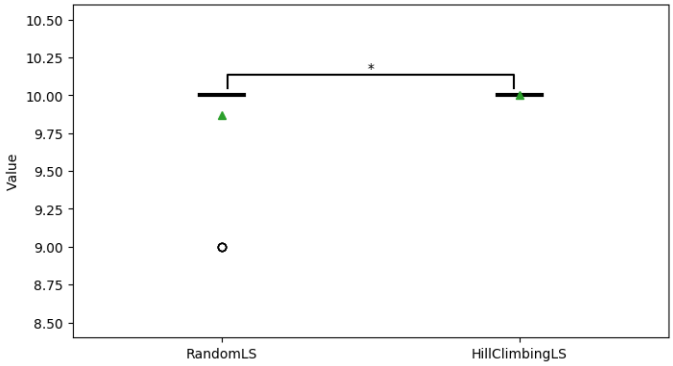


Taille - large

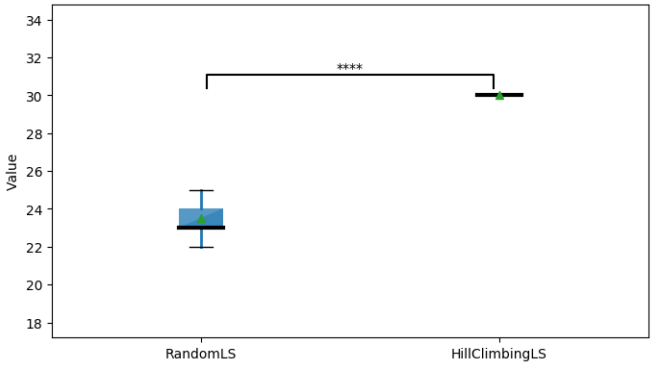


Test – One Max

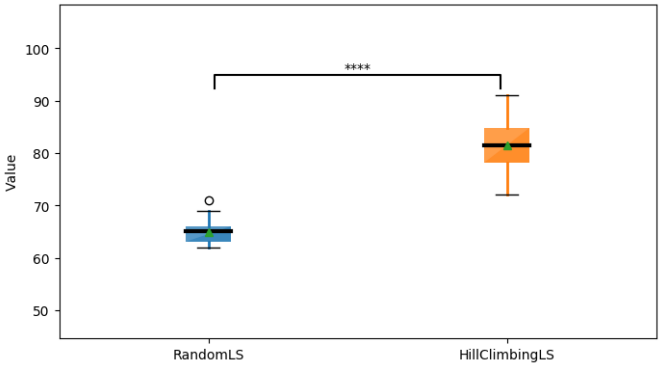
Taille – small



Taille – medium

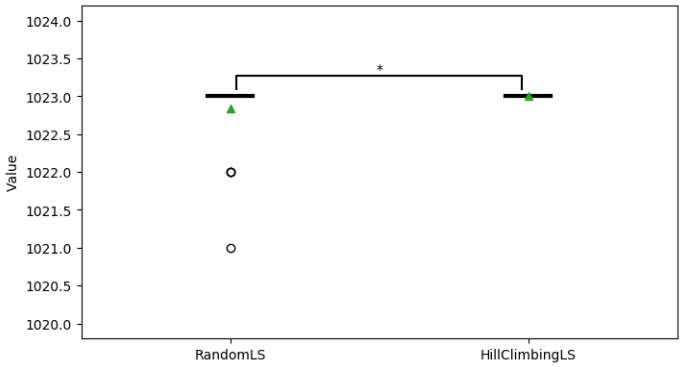


Taille - large

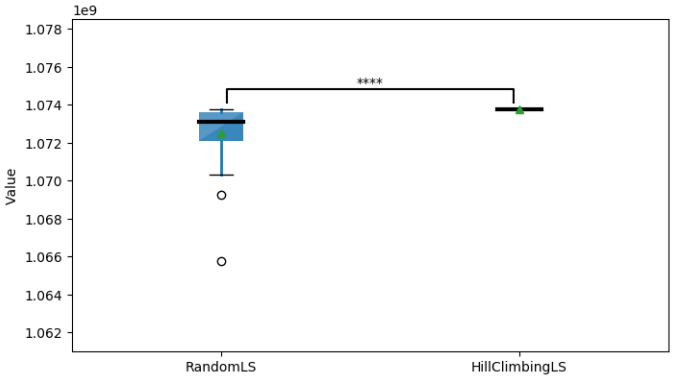


Test – Binval

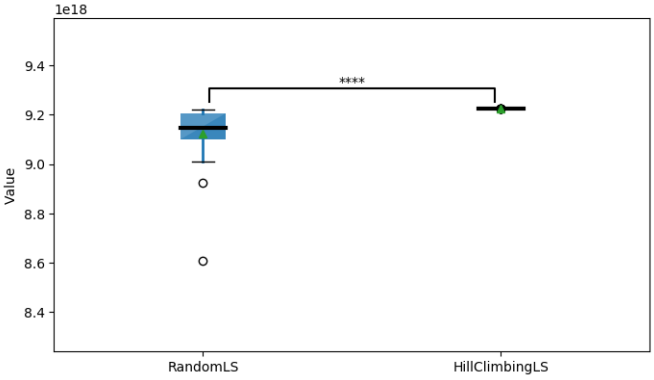
Taille – small



Taille – medium

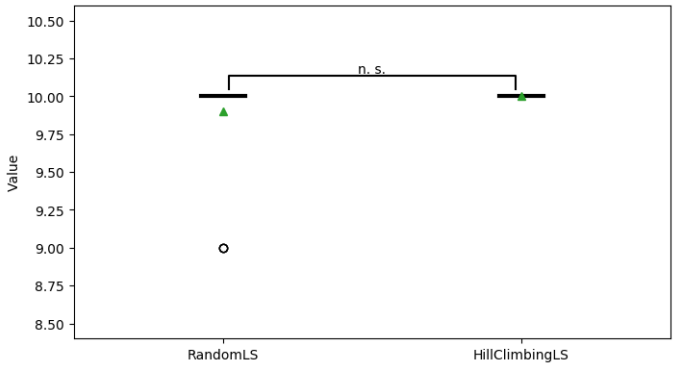


Taille - large

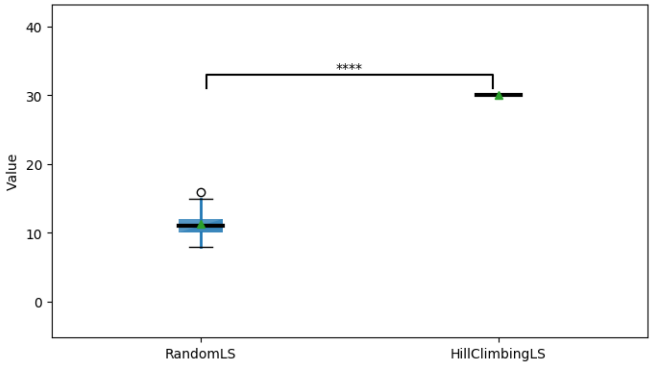


Test – Leading Ones

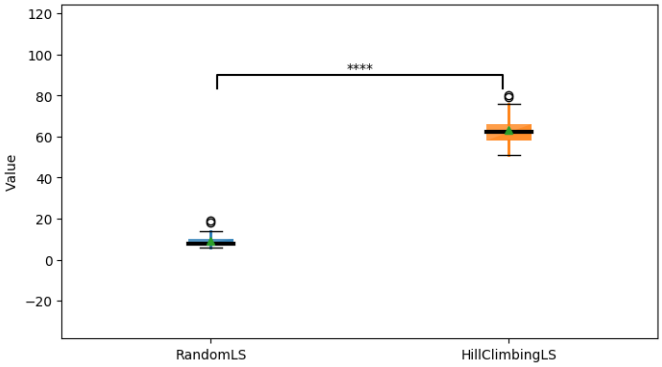
Taille – small



Taille – medium



Taille - large



Commentaire

On constate que l'algorithme RandomLS se comporte plutôt bien sur des petites instances de problèmes car il a le temps d'explorer une grande partie de l'espace des solutions (si ce n'est la totalité) et donc il est efficace.

A noter cependant que pour des problèmes de type voyageur du commerce (TSP) l'algorithme RandomLS a tout le temps des difficultés, difficulté sans doute lié à la grande taille de l'espace de recherche.

On notera également que HillClimbingLS a des difficultés pour le problème du sac à dos, en effet celui-ci ne peut retirer aucun objet ! C'est sans doute pour cette raison qu'il moins performant dans ce type de problème.

Code commenté de HillClimbingLS

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
import random
from generic import *

class HillClimbingLS(LocalSearchAlgorithm):

    def __init__(self, prob, options):
        """ Constructeur de la super classe
        """
        LocalSearchAlgorithm.__init__(self, prob, options)

    def get_neighbors(self):
        """ retourner les voisins de la solution courante
        """
        return self._solution.neighbors()

    def filter_neighbors(self, neighbors):
        """ filtrer toutes les solutions violant les contraintes
        """
        return [ n for n in neighbors if self._problem.feasable(n) ]

    def select_next_solution(self, candidates):
        """ Si il y des solutions (après filtrage), retourner la meilleure
        """
        best_candidat = candidates[0]
        best_solution = self._problem.eval(best_candidat)

        # On parcourt toute les solutions
        for candidat in candidates[1:]:
            # On évalue les solutions parcourues
            current_solution = self._problem.eval(candidat)
            # Si la solution est meilleure que la solution courante, on sauvegarde la solution (le
candidat) et sa valeur (best_solution)
            if self.better(current_solution,best_solution):
                best_solution = current_solution
                best_candidat = candidat
        return best_candidat

    def accept(self, new_solution) :
        """ HillClimbingLS accepte seulement les solutions qui sont meilleures que la solution
courante"""

        cur_val = self._solution.value
        new_val = new_solution.value
        # On prend la meilleure solution (better tient compte du fait que le problème soit
minimisation ou maximisation)
        return self.better(new_val,cur_val)
```