TP 2 – Métaheuristique

Bellanger Clément

Résultats

Les tests on été effectué sur les trois problèmes principaux : Sac à dos set covering et voyageur de commerce (TSP)

Pour les test, suite à un bug d'affichage, on a divisé les tests en deux parties :

Partie 1 : On compare RandomLS à HillClimbingLS, RandomizedHillClimbingLS et FirstImprovementHillClimbingLS

Partie 2 : On compare RandomLS à SimulatedAnnealingLS, et TabuLS

Les tests ont étés effectués avec les paramètres suivants :

SimulatedAnnealingLS

Température initiale : 300

gamma : 0.99

RandomizedHillClimbingLS

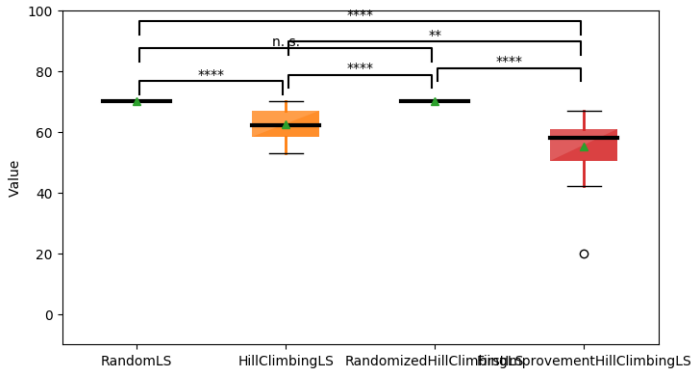
alpha : 0.8

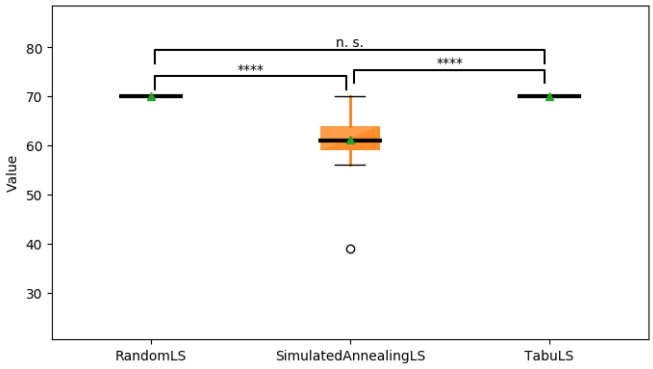
TabuLS

taille de la liste : 50

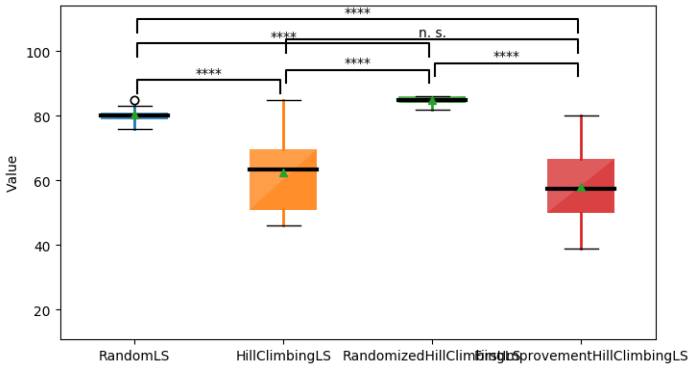
Problème – Sac à Dos

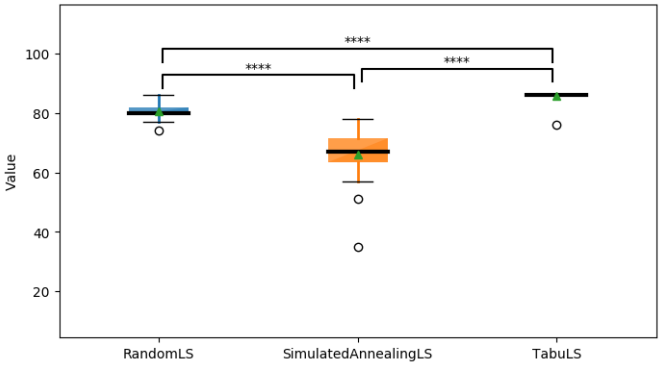
Taille – small



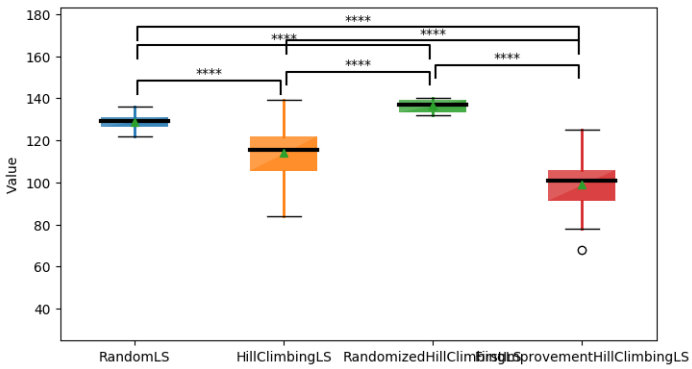


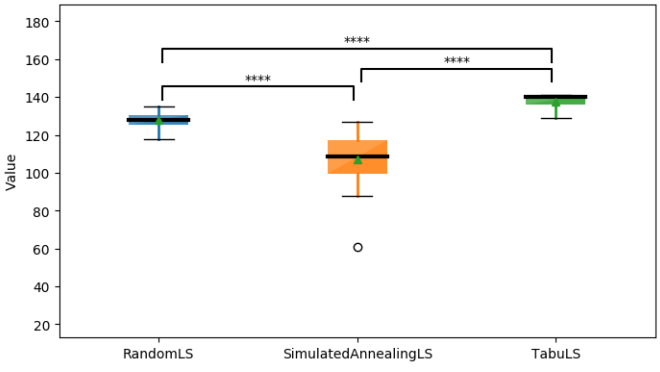
Taille – medium





Taille – large

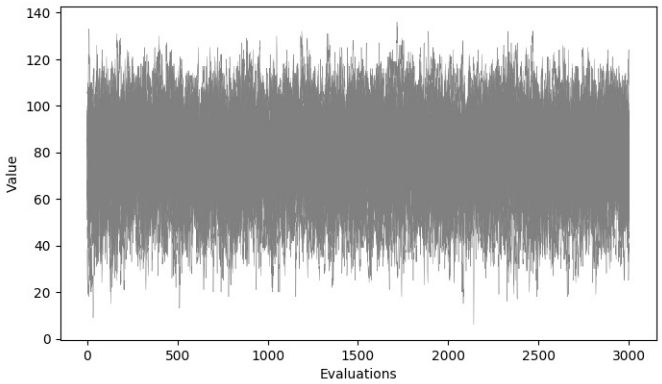




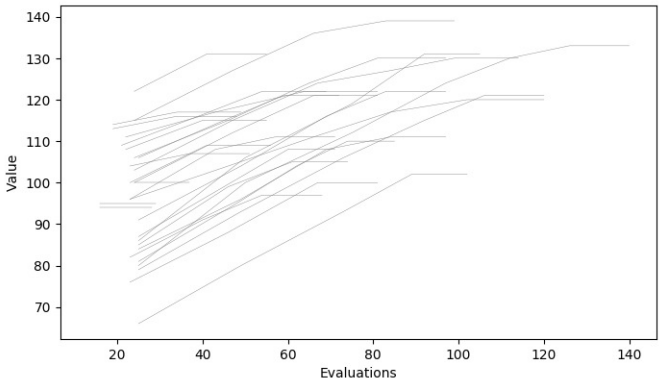
Problème – Sac à Dos

Trace des algorithmes (taille large)

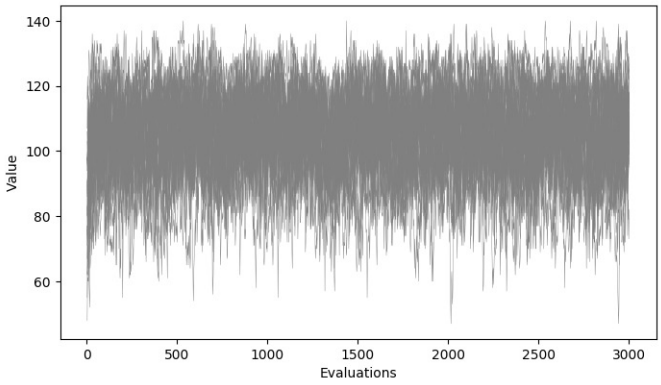
RandomLS



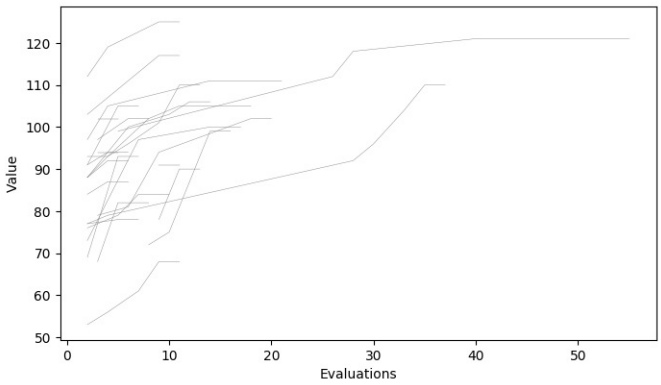
HillClimbingLS



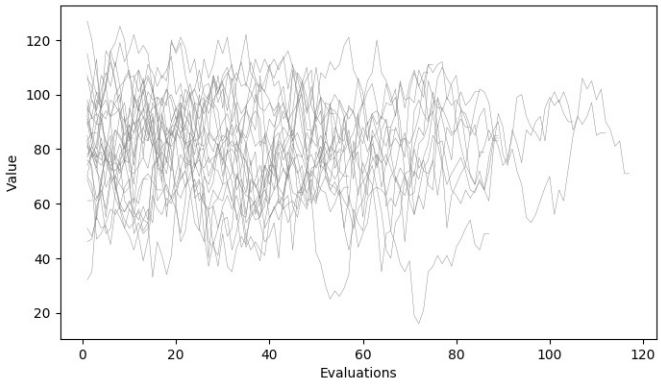
RandomizedHillClimbingLS



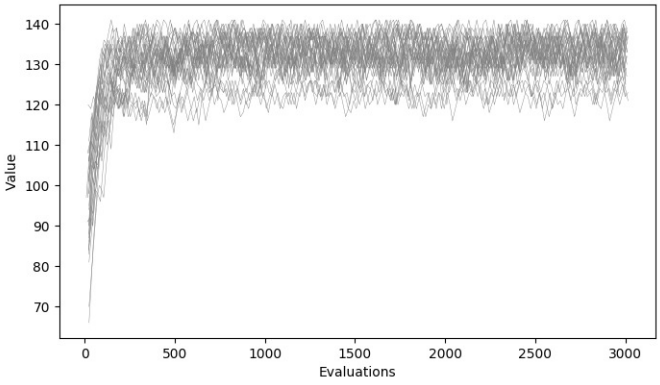
FirstImprovementHillClimbingLS



SimulatedAnnealingLS

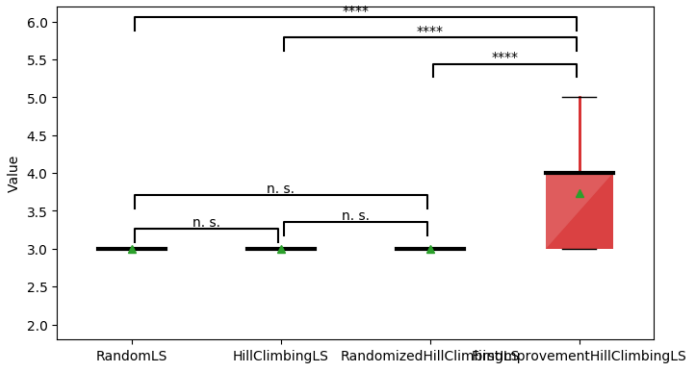


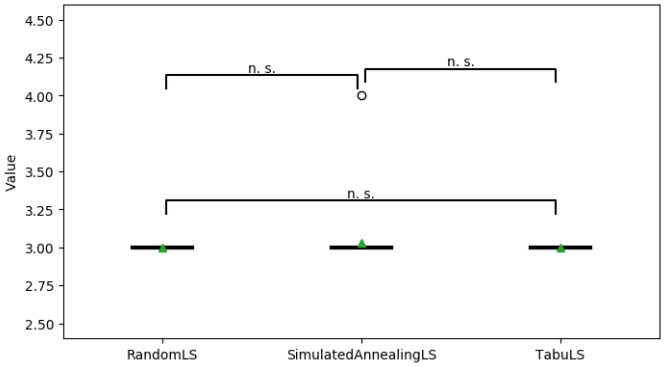
TabuLS



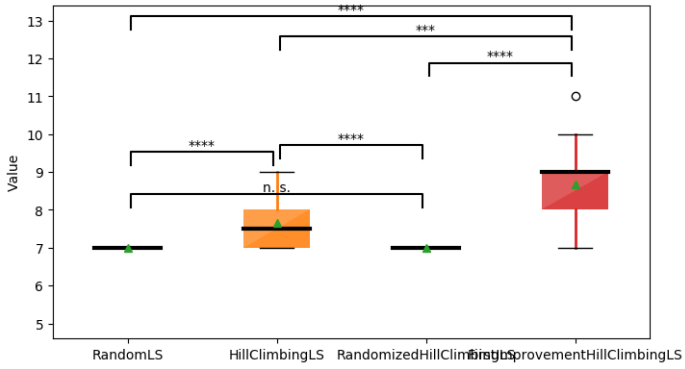
Problème – Set Covering

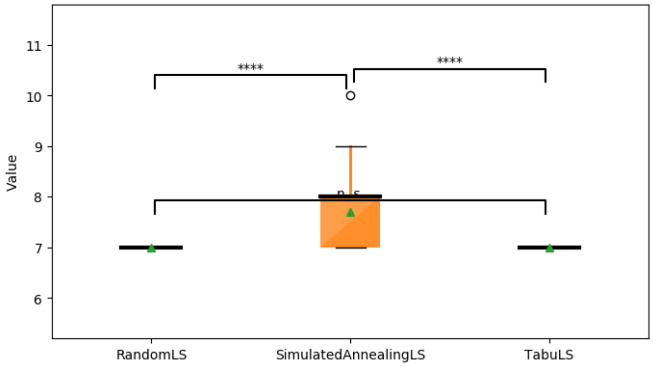
Taille – small



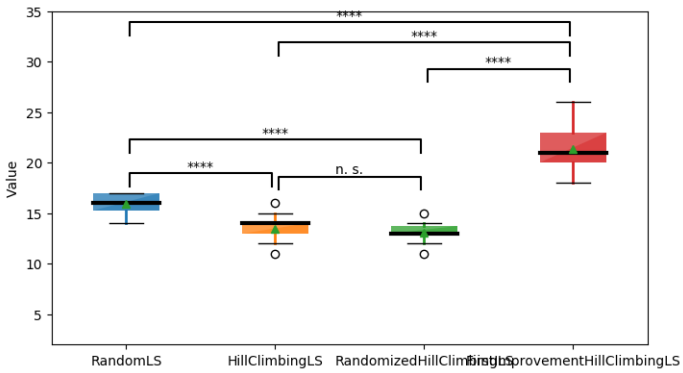


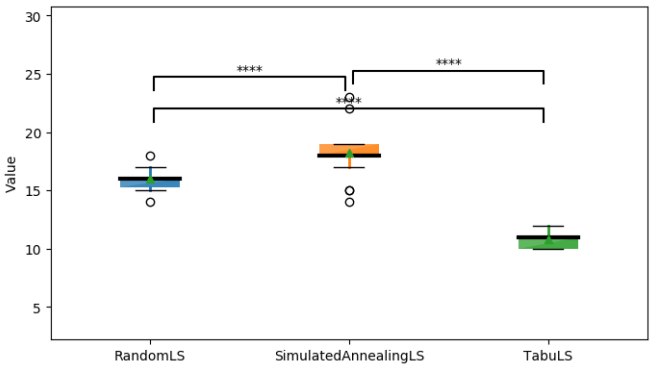
Taille – medium





Taille – large

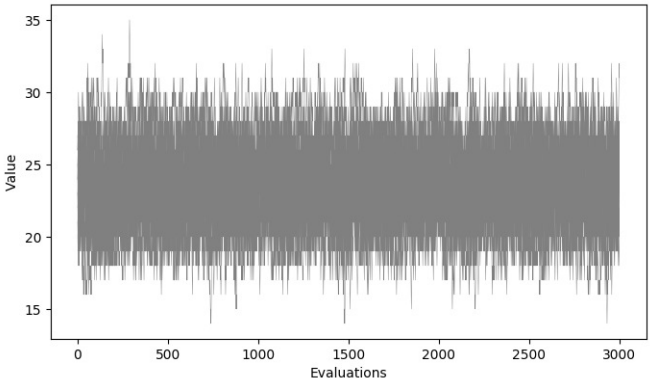




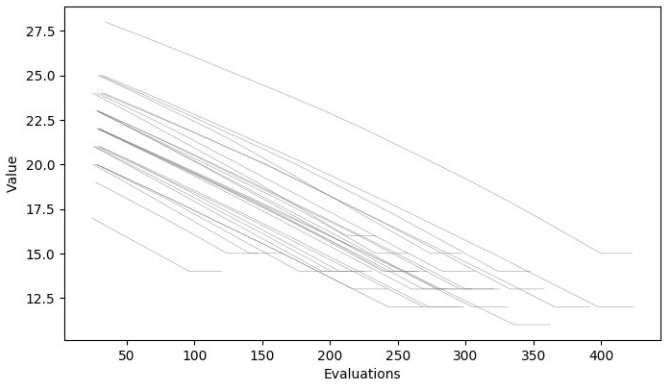
Problème – Set Covering

Trace des algorithmes (taille large)

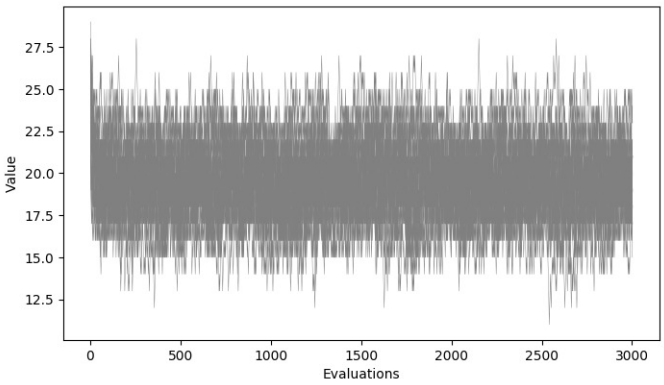
RandomLS



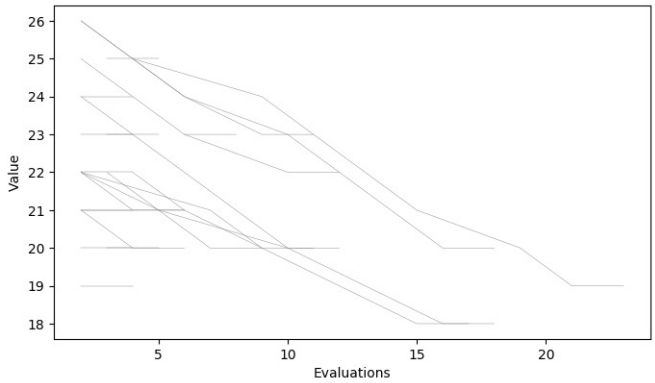
HillClimbingLS



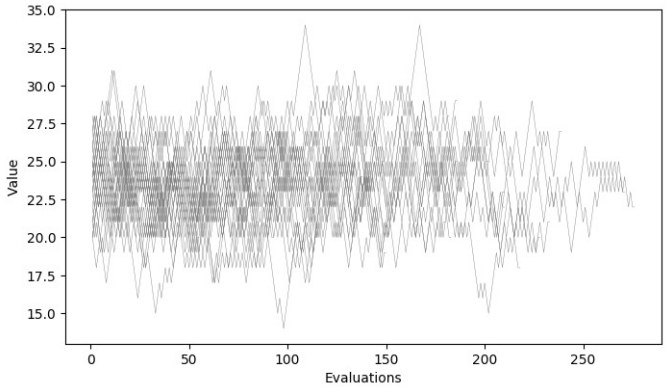
RandomizedHillClimbingLS



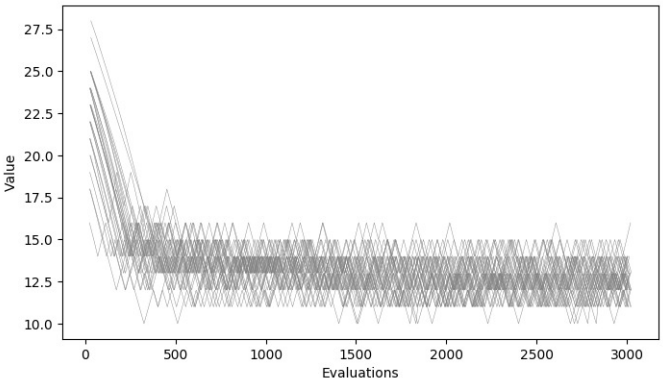
FirstImprovementHillClimbingLS



SimulatedAnnealingLS

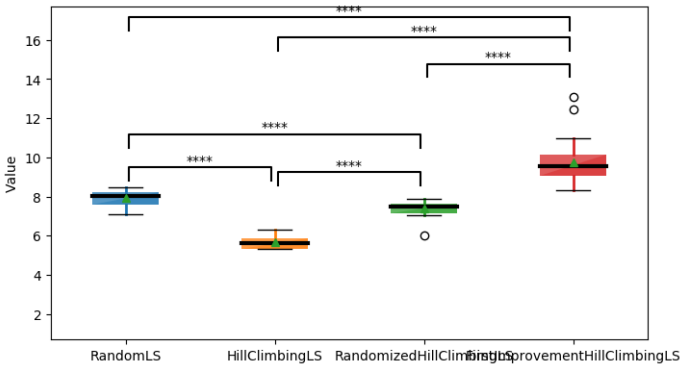


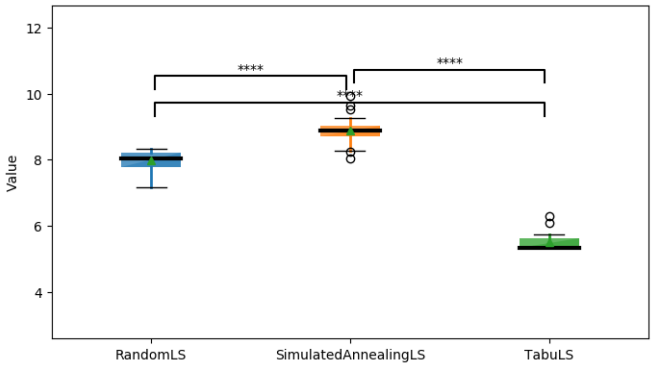
TabuLS



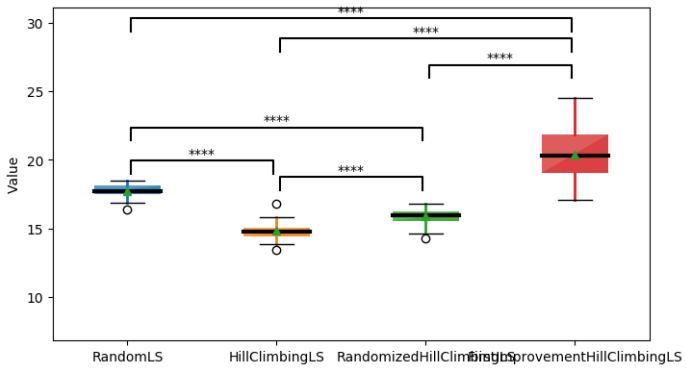
Problème – Voyageur de commerce (TSP)

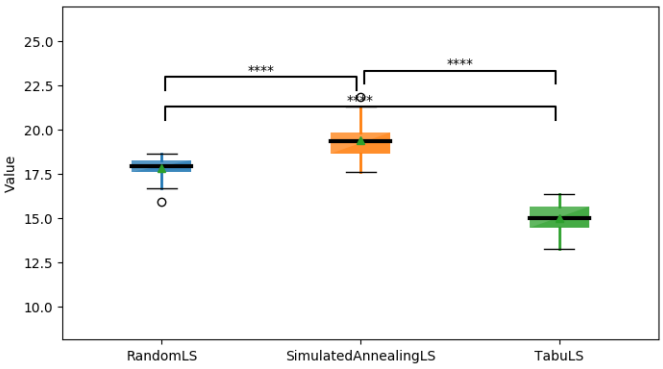
Taille – small



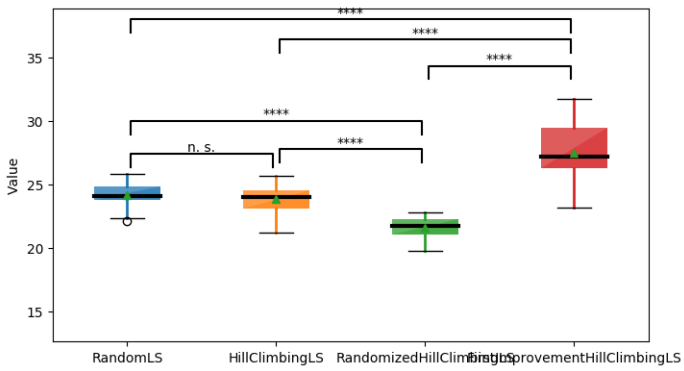


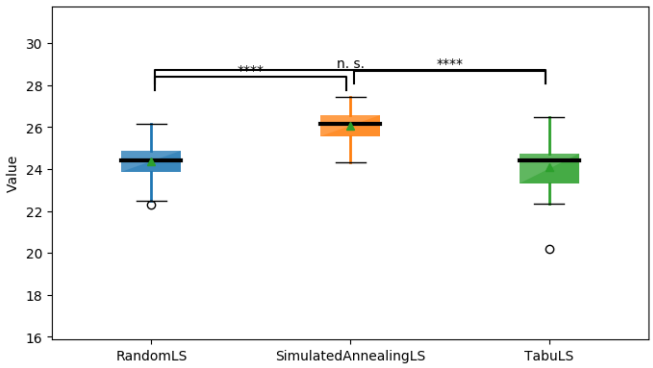
Taille – medium





Taille - large

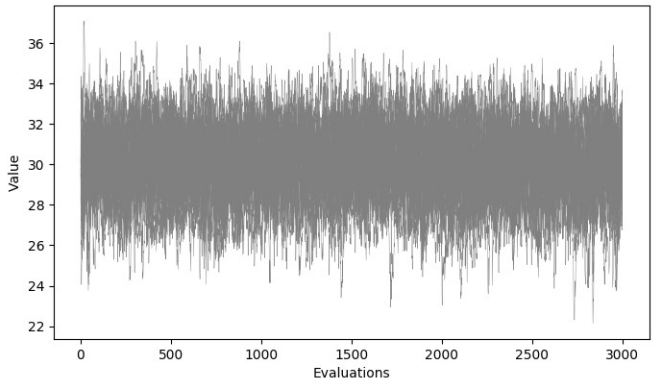




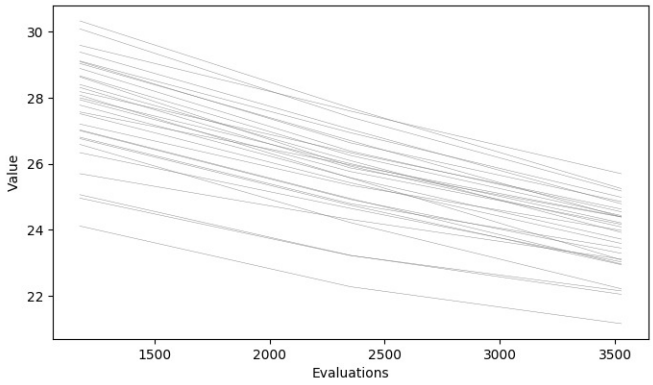
Problème – Voyageur de commerce (TSP)

Trace des algorithmes (taille large)

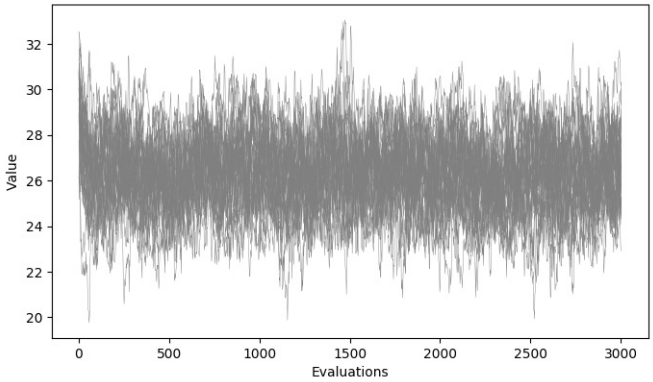
RandomLS



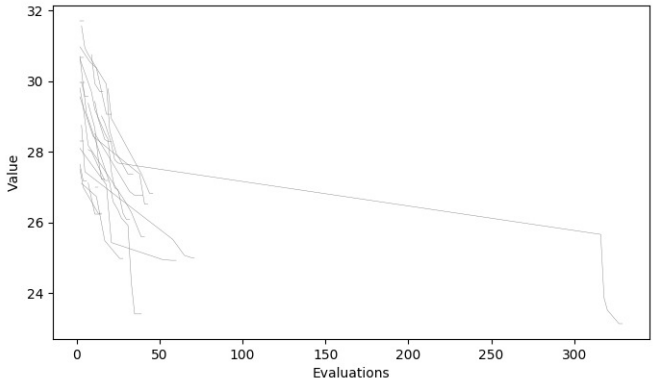
HillClimbingLS



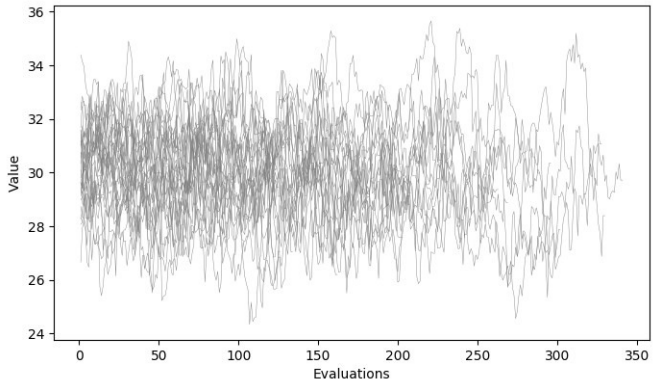
RandomizedHillClimbingLS



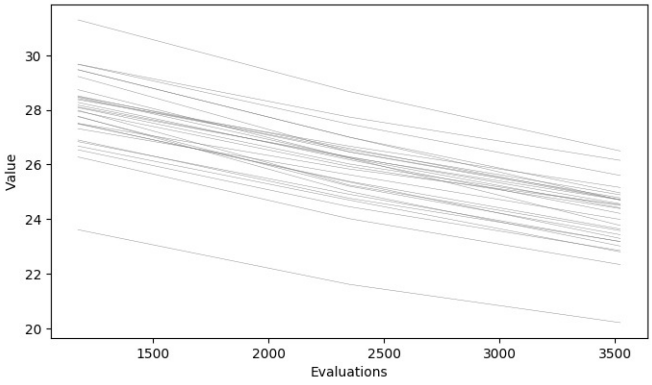
FirstImprovementHillClimbingLS



SimulatedAnnealingLS



TabuLS



Commentaire

On constate que pour les problèmes de type sac à dos les algorithmes acceptant toujours les solutions réussissent mieux.

En particulier la recherche taboue donne les meilleurs résultat pour ce type de problème.

Lorsque l'on modifie avec le paramètre gamma de l'algorithme de recuit simulé, on constate que les meilleurs résultats surviennent avec un gamma élevé (proche de 1) et une température initiale haute (> 200). En effet il faut une température haute pour être initialement dans une phase exploratoire et un gamma élevé permet de garder une certaine inertie et rester plus longtemps dans cette phase exploratoire.

Après des tests supplémentaires je constate que la valeur de alpha est sans doute trop basse pour le RandomizedHillClimbing, en effet le comportement se rapproche trop du hasard avec une valeur de 0,5 l'algo n'a pas vraiment le temps d'améliorer ses solutions.

Code commenté

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

# Structure temporaire à améliorer
alpha=0.8
class RandomizedHillClimbingLS(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 génère une probabilité entre 0 et 1 et on regarde si elle est supérieure à alpha si c'est le cas on fait du hill_climbing
        # Sinon on prend un voisin aléatoire
        if (random.uniform(0, 1) <= alpha) :
            # 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
        # On sélectionne un voisin aléatoire si il existe (que l'on prend soin d'évaluer avant)
        if len(candidates) > 0 :
            candidate_random=random.choice(candidates)
            self._problem.eval(candidate_random)
            return candidate_random
        return None


    def accept(self, new_solution) :
        """ RandomizedHillClimbingLS accepte toujours la solution"""

        #cur_val = self._solution.value
        #new_val = self._problem.eval(new_solution)
        return True
```

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

class FirstImprovementHillClimbingLS(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 première meilleure
        """
        first_candidat = candidates[0]
        best_solution = self._problem.eval(first_candidat)

        for candidat in candidates[1:]:
            current_solution = self._problem.eval(candidat)
            # Dès qu'une solution est meilleure que la solution actuelle, on la retourne
            if self.better(current_solution,best_solution):
                better_candidat = candidat
                return better_candidat
        return first_candidat

    def accept(self, new_solution) :
        """ FirstImprovementHillClimbingLS accepte la solution seulement si elle est meilleure.
        """
        cur_val = self._solution.value
        new_val = new_solution.value
        return self.better(new_val,cur_val)
        #return True
```

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

# Structure temporaire à améliorer
# Température initiale
initalTemp=300
# Coefficient de baisse de température
gamma=0.99
class SimulatedAnnealingLS(LocalSearchAlgorithm):

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

    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 une solution au hasard
        """
        # On sélectionne un voisin aléatoire si il existe (que l'on prend soin d'évaluer avant)
        if len(candidates) > 0 :
            candidate_random=random.choice(candidates)
            self._problem.eval(candidate_random)
            return candidate_random
        return None


    def accept(self, new_solution) :
        """ SimulatedAnnealingLS accepte toujours les solutions meilleures mais seulement avec une
probabilité alpha les solutions moins bonnes"""

        # Soit la solution est meilleure auquel cas on l'accepte toujours
        if self.better(new_solution.value,self._solution.value) :
            # Mise à jour de la température
            self.temperature=self.temperature*gamma
            return True
        # Soit la solution est moins bonne et dans ce cas on a une probabilité de l'accepter dépendant
de l'aléatoire et de la température
        else :
            alpha=math.exp(-abs(self._solution.value-new_solution.value)/self.temperature)
            # Mise à jour de la température
            self.temperature=self.temperature*gamma
            if (alpha > random.uniform(0, 1)) :
                return True
            else :
                return False
        #cur_val = self._solution.value
        #new_val = self._problem.eval(new_solution)
        return True
```

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

# Structure temporaire à améliorer
# Taille de la liste taboue
tabuSize=50
class TabuLS(LocalSearchAlgorithm):

    def __init__(self, prob, options):
        """ Constructeur de la super classe
        """
        LocalSearchAlgorithm.__init__(self, prob, options)
        self.tabuList=deque(maxlen=tabuSize)

    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 qui n'est pas dans la liste taboue
        """
        # Au départ aucune solution valide n'est trouvée
        best_candidat=None

        # On parcourt toute les solutions
        for candidat in candidates[1:]:
            # Si il n'est pas dans la liste taboue
            if candidat not in self.tabuList :
                # Si on a pas encore de candidat valide
                if best_candidat is None :
                    best_candidat = candidat
                    best_solution = self._problem.eval(best_candidat)
                else :
                    # 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
        # On retourne le candidat qu'il soit nul ou non
        return best_candidat

    def accept(self, new_solution) :
        """ TabuLS accepte toujours la solution"""

        # Si la solution n'est pas nulle on l'ajoute à la liste taboue
        if new_solution is not None:

            # Si la taille de la liste taboue est la taille maximale-1 alors on supprime le premier élément (FIFO)
            if (len(self.tabuList) == self.tabuList.maxlen-1) :
                self.tabuList.popleft()

            self.tabuList.append(new_solution)

        return True
```