UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE FLORESTAL
CIÊNCIA DA COMPUTAÇÃO

HEITOR PASSEADO SOARES(3055) E VITOR HUGO DE OLIVEIRA(3049)

META-HEURÍSTICAS

TRABALHO PRÁTICO 1

FLORESTAL, MG
05 de Setembro de 2021

HEITOR PASSEADO SOARES(3055) E VITOR HUGO DE OLIVEIRA(3049)

META-HEURÍSTICAS
TRABALHO PRÁTICO 1

Documentação do 1º trabalho prático de Meta-heurísticas que tem como objetivo o desenvolvimento e a aplicação de dois algoritmos de meta-heurísticas

Orientador: Prof. Dr. Marcus Henrique Soares Mendes

FLORESTAL, MG

05 de Setembro de 2021

Sumário

1 Introdução

Este trabalho consiste na implementação e aplicação do algoritmo de Hill-Climbing e Iterated Local Search, comparando as suas performances e resultados obtidos com 30 iterações para cada algoritmo.

2 Desenvolvimento

2.1 Decisões

2.1.1 Variáveis de decisão

Foi realizado a criação de uma classe a qual representasse as variáveis de decisão, afim de facilitar usabilidade destas no restante do algorítimo, nesta classe é armazenado a sua label correspondente(x,y,z...), o limite inferior o limite superior, valor atual e para o algorítimo do hill-climbing adicionamos também o valor aleatório de sua última modificação. Uma facilidade traga devido a esta escolha de implementação é que adicionar novas variáveis de decisão de acordo com a equação objetivo a ser usada não resultaria em grande ônus. Por fim, também criamos uma função para realizar a cópia da lista de variáveis de decisões.

```python
class Variable:
    def __init__(self, label, li, ls, ri=1, value: float = 0):
        self.label = label
        self.li = li  # Limit inferior
        self.ls = ls  # Limit superior
        self.ri = ri  # Valor aleatorio de 0 a 1
        self.value = value

    def __str__(self):
        return "{:.3f}".format(self.value)


def copyList(original: List[Variable]):
    return [Variable(obj.label, obj.li, obj.ls, obj.ri, obj.value) for obj in original]
```

Figura 1 – Classe Variable

2.1.2 Heurística ISL

A pertubação do Iterated Local Search era uma randomização completa entre o limite inferior e o limite superior, o critério para decidir se usaríamos as variáveis perturbadas é definido pela função "criteria"

```python
def criteria(objective, s_old: List[Variable], s_new: List[Variable], memory: MemoryILS):
    """
    @param objective:
    @param s_old: Varaiveis de decisão antigo
    @param s_new: Varaiveis de decisão apos pertubação e busca local
    @param memory:
    """

    value_s_new = objective(*s_new)

    # Se for melhor que todos aramazena em memoria
    if objective(*memory.best_variables) > value_s_new:
        memory.best_variables = s_new
        return s_new  # Explotar

    # Se for pior que todos na memoria e for pior que o valor objetivo de s1
    if memory.isWorstThanAllMemory(value_s_new) and value_s_new > objective(*s_old):
        if memory.acceptanceProbability() > random.random():
            return s_new
        else:
            return s_old
    else:
        return s_old
```

Figura 2 – Função critéria

Se as variáveis perturbadas dão um resultado melhor, isso é armazenado na memória e o ISL continua iterando, se é um resultado mediano as variáveis permanecem inalteradas e caso dê um resultado pior que todos os outros, pode indicar uma nova bacia de atração. Nesse caso joga-se uma "moeda"para ver se o algoritmo irá aderir à essa nova bacia e priorizar exploração em vez de explotação. A probabilidade de se mudar de bacia diminui a medida que o número de iterações aumenta.

2.1.3 Hill Climbing

O nosso algoritmo do hill climbing foi feito com p = 1 por convenção como estudado na disciplina e fizemos o hill climbing mais íngreme o que implica em ter mais iterações em um determinado pontos de variáveis de decisão o que deixa o resultado mais preciso se ele acertar a melhor bacia de decisão, ele prioriza explotação em vez de exploração. Outra decisão tomada foi quanto a modificação realizada nas variáveis de decisão, para essa parte nós baseamos no algoritmo da bibliografia base, de forma a gera um valor aleatório que caso esteja dentro de um valor P aceitável, este e somado ao limite inferior e multiplicado com o limite superior menos o inferior, assim resultando em um valor aleatório entre o limite inferior e superior.

```python
def modify(candidates: List[Variable], p):
    for candidate in candidates:
        if p >= candidate.ri:
            while True:
                _, candidate.ri = noise(candidate.li, candidate.ls)
                _, n = noise(-candidate.ri, candidate.ri)

                if candidate.li <= (candidate.value + n) <= candidate.ls:
                    break
            candidate.value = candidate.value + n
    return candidates
```

Figura 3 – Código de modify

2.1.4 Constantes

- HILL_STEEPER_ITERATIONS: Número de modificações no Hill Climbing mais ingrime = 100
- HILL_ITERATIONS: Número máximo para parada do Hill Climbing = 300
- N_ITERATIONS: Número máximo de interações ILS = 300
- BASIS_LIMIT: Numero para determinar se mudou de bacia ou não = 30

2.2 Hill-Climbing

A função recebe a equação a qual iremos analisar, um vetor de variáveis de decisões, um valor P de aceitabilidade do tamanho da aleatoriedade, e o número de iterações do algor imito como condição de parada. Assim o algoritimo inicia o vetor de variáveis de decisão com valores aleatórios dentro dos limites de cada variável e enquanto o limite de iterações sem melhoria não é alcançado, ele continua ajustando e salvando em uma cópia r e comparando as soluções de r e s para saber qual o melhor.

```python
def hill_climbing(objective: Callable, variables: List[Variable], p=P,
                  n_iterations=HILL_ITERATIONS, show_result=False):
    s = initialize(variables)
    it_count = 0

    while True:
        it_count += 1

        r = modify(copyList(s), p)

        for i in range(HILL_STEEPER_ITERATIONS):
            w = modify(copyList(s), p)
            if quality(w, objective) >= quality(r, objective):
                r = w

        if quality(r, objective) > quality(s, objective):
            s = r

        if isIdeal(s) or it_count >= n_iterations:
            if show_result:
                print(f"Resultado: ({s[0]},{s[1]}) = {objective(*s)}")
            break

    return s

```

Figura 4 – Função HC

2.3 Iterated Local Search

A função recebe a equação a qual iremos analisar, um vetor de variáveis de decisões, uma função para busca local, caso contrario se utiliza do próprio hill-clibing explicado na seção acima. O algorítimo inicia criando a variavel memory que será responsável por armazenar o histórico dos resultados já encontrados, a melhor lista de variáveis de decisão até então encontrados e o número de iteração vigente no algoritmo, nesta classe também realizamos a criação de diversas funções auxiliares as quais foram uteis para construção do algoritmo ILS. Apos inicializar a variável memory, inicializamos s0 com uma solução inicial aleatória, a qual posteriormente e realizada uma busca local na mesma que será armazenada em s1 e assim setamos o melhor solução até então encontrada como s1.

Em sequência entramos na parte iterativa do algorítimo ao qual realizamos uma pertubação na solução s1, armazenando-a em s2, neste ponto e realizado uma pertubação na solução atual s1 de forma a sair da bacia atual, está e feita comparando com os valores no histórico de forma que a nova solução esteja ao menos BASIS_LIMIT de distância de

todas as outras soluções do histórico até então encontradas. O próximo passo é realizar a busca local em s2, como dito anteriormente está busca local pode ser através de uma passada nós paramentos ou como default será o hill-climbing em nosso trabalho todos os resultado mostrado em sequência se utilizaram do valor default, o resultado desta busca local é armazenada em s3. Em fim, chegamos ao passo do critério de aceitação ao qual avaliamos se s1 ou s3 valem mais apena de ser explorados. se optemos por s1 estaremos explotando e por s3 estaremos explorando. A parte do critério de aceitação e explicado na seção acima.

```python
def ILSearch(objective: Callable, variables: List[Variable], local_search=None):
    memory = MemoryILS()
    local_search = local_search or hill_climbing

    s0 = initialize(variables)  # Gerar solução inicial s0
    memory.best_variables = s0
    s1 = local_search(objective, s0)  # Busca local em s0

    while True:
        memory.increaseIteration()

        s2 = perturbation(objective, copyList(s1), memory)
        s3 = local_search(objective, s2)
        s1 = criteria(objective, s1, s3, memory)

        if memory.terminated():
            break

    return s1
```

Figura 5 – Função ILS

3 Resultados obtidos

3.1 Problema 1

Função: f(x)= sin(x+y)+(x-y)² - 1.5x + 2.5y + 1

3.1.1 A)

Regras de entrada: 1.5 x 4 e 3 y 4

Algoritmo	Mínimo	Máximo	Média	desvio padrão
HC	-1.91292	2.27182	-0.81352	1.27718
ILS	-1.91322	-1.91318	-1.91322	9.46061e-06

Problema 1 com -1.5 x 4 e 3 y 4

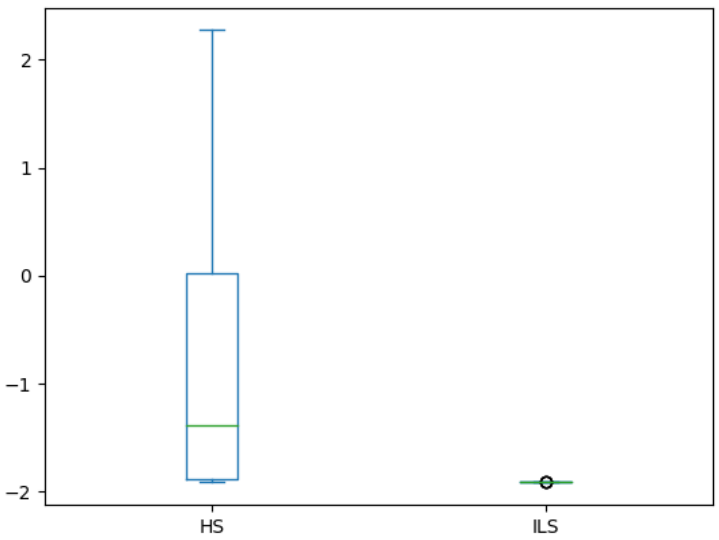

Figura 6 – Função BoxPlot Função 1 com -1.5 x 4 e 3 y 4

Podemos notar que apesar hill climbing e o ILS terem valores muito iguais o ILS foi capaz de resultar em uma melhora em relação ao hill climbing com desvio padrão muito inferior.

3.1.2 B)

Regras de entrada: 1 x 0 e 2 y -1

Algoritmo	Mínimo	Máximo	Média	desvio padrão
HC	-1.91322	-1.91322	-1.91322	3.76283e-06
ILS	-1.91322	-1.91322	-1.91322	1.14733e-07

Problema 1 com 1 x 0 e 2 y -1

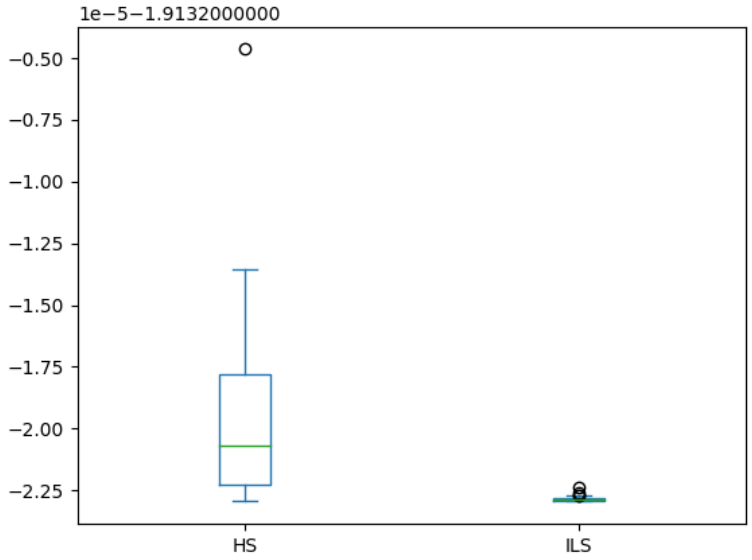

Figura 7 – Função BoxPlot Função 1 com 1 x 0 e 2 y -1

Nessa análise, podemos notar que hill climbing e o ILS tiveram valores idênticos o que implica que ambos os algoritmos conseguiram chegar no mínimo global. Diferente do anterior o rio crime obteve melhor desvio padrão.

3.2 Problema 2

3.2.1 A)

Algoritmo	Mínimo	Máximo	Média	desvio padrão
HC	-931.648	-427.398	-713.897	121.75
ILS	-959.523	-934.051	-953.084	6.16745

Problema 2 com -512 <= x, y <= 512

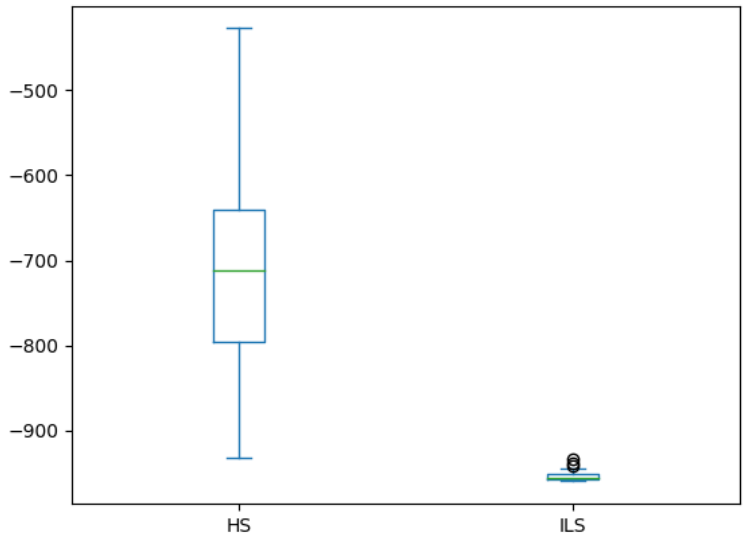

Figura 8 – Função BoxPlot Função 1 com 1 x 0 e 2 y -1

No meu Problema 2 é possível notar claramente a melhora do ILS em relação ao HC que além de conterem desvios padrões muitos diferentes, ILS com o melhor neste caso o ILS ainda assim também conseguiu obter o melhor resulto.

3.2.2 B)

Algoritmo	Mínimo	Máximo	Média	desvio padrão
HC	-959.638	-959.364	-959.563	0.0552227
ILS	-959.64	-959.63	-959.637	0.002812

Problema 2 com limites [511, 512] para x e [404, 405] para y

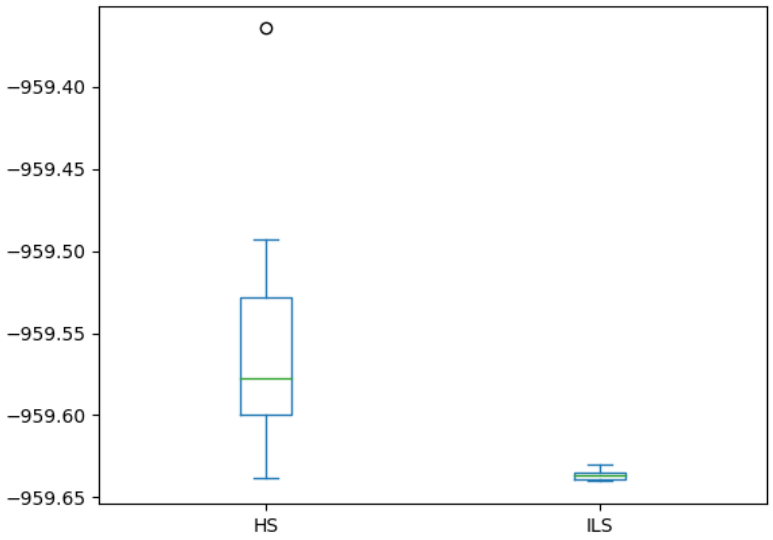

Figura 9 – Função BoxPlot Função 1 com 1 x 0 e 2 y -1

Nesse último resultado acontece algo de muito interessante onde podemos notar que ambos os algoritmos provavelmente devem ter ficado preso em alguma solução local uma vez que ambos possuem valores muito semelhantes.

4 Conclusão

Podemos concluir que ao longo deste trabalho foi possível observar o funcionamento do Hill-climbing e o como o do ILS junto ao Hill-Climbing e assim averiguar e reafirmar os conhecimentos vistos em aula. Além disso, podemos notar a diferença a qual as mudanças de parâmetros realizam nos algoritmos e como e de grande importância realizar a construção de um código modular que facilite a alteração destas para chegar ao melhor ótimo possível de acordo com cada função.

5 Referências

Aula 07 - ILS. Disponível em: https://www2.cead.ufv.br/sistemas/pvanet /files/-conteudo/5546/AULA04R.pdf Acesso em: 3 de setembro de 2021.

Aula 04 - Hill-climbing. Disponível em: https://www2.cead.ufv.br/sistemas/pvanet /files/conteudo/5546/AULA04R.pdf Acesso em: 3 de setembro de 2021.