PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
CIÊNCIA DA COMPUTAÇÃO

MARCOS RODOLFO CRUVINEL GOULART QUERINO

IMPLEMENTAÇÃO EM PYTHON DA META HEURISTICA BUSCA TABU PARA RESOLVER O PROBLEMA ONEMAX

GOIÂNIA
2020

Teste do programa com os seguintes dados de entrada e sua respectiva resposta:

- Entrada: 4 bits, BTmax= 1 e T= 1. Resultado:

"Solucao inicial: 0011
Avaliacao da solucao inicial: 2

Total de iteracoes: 4
Melhor iteracao: 2
Melhor solucao: 1111
Avaliacao da melhor solucao: 4"

- Entrada: 10 bits, BTmax= 2 e T= 2. Resultado:

"Solucao inicial: 0010110010
Avaliacao da solucao inicial: 4

Total de iteracoes: 9
Melhor iteracao: 6
Melhor solucao: 1111111111
Avaliacao da melhor solucao: 10"

- Entrada: 50 bits, BTmax= 2 e T= 1. Resultado:

"Solucao inicial: 11011010110000010000100001000000110100011100011111
Avaliacao da solucao inicial: 21

Total de iteracoes: 32
Melhor iteracao: 29
Melhor solucao: 11111111111111111111111111111111111111111111111111
Avaliacao da melhor solucao: 50"

- Entrada: 100 bits, BTmax= 2 e T= 2. Resultado:

“Solucao inicial:
1111001100001100000110011000110101011001001101100100110110001100101100001101111000010000111111111101
Avaliacao da solucao inicial: 51

Total de iteracoes: 52
Melhor iteracao: 49
Melhor solucao:
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
Avaliacao da melhor solucao: 100”

Implementação do programa:

```
#chamando o modulo random para usar sua funcao randint (numeros inteiros aleatorios)
from random import randint

# funcao para gerar aleatoriamente uma solucao inicial qualquer
def solucao_inicial(tam): # "tam" é o tamanho em bits da solucao inicial
    sol= [str(randint(0,1)) for i in range(tam)] # gera uma string com um conjunto aleatorio entre 0 e 1
                                                 # "for" preenche os bits dessa string
    return ''.join(sol) # retorna essa string criada aleatoriamente

# funcao de avaliacao de uma solucao (retorna a quantidade de '1')
def avaliacao(sol):
    return sol.count('1') # retorna a soma dos caracteres '1' na string 'sol'
```

```
# funcao geradora de vizinhos
def gerar_vizinhos(sol):
    vizinhos= [] # lista de vizinhos vazia
    for i in range(len(sol)): # percorre a string bit a bit
        lista_bits= list(sol) # transforma a string "sol" em lista

        # realizar o movimento 'm' para trocar o bit
        lista_bits[i]= '1' if sol[i]=='0' else '0' # se o bit for '0', troca para '1', senao, troca para '0'

        vizinho= ''.join(lista_bits) # transformar de volta para string
        vizinhos.append(vizinho) # e adiciona na lista de vizinhos
    return vizinhos # depois de gerar os vizinhos de 'sol' retorna essa lista

# funcao que retorna o bit que foi alterado no vizinho
def movimento_tabu(sol, vizinho): # compara 'sol' com 'vizinho'
    for i in range(len(sol)): # percorre a string 'sol'
        if sol[i] != vizinho[i]: # se uma posicao de 'sol' for diferente da de 'vizinho'
            return i # retorna essa posicao

# funcao para encontrar o melhor vizinho nao tabu
# se a melhor solucao for tabu, testaremos o criterio de aspiracao por objetivo
def obter_melhor_vizinho(vizinhos, lista_tabu, melhor_sol, sol): # parametros: lista de vizinhos
                                                                 # lista de posicoes de movimentos proibidos
                                                                 # melhor solucao encontrada ate o momento
                                                                 # solucao corrente
    # primeiro, vamos ordenar a lista de vizinhos de forma decrescente, da maior para menor avaliacao
    vizinhos.sort(key= avaliacao, reverse=True) # key = criterio de avaliacao da ordenacao e reverse= decrescente
```

```
    # percorrer a lista de vizinhos
    for vizinho in vizinhos:
        # verificar se o movimento que gerou o vizinho esta na lista tabu
        if movimento_tabu(sol, vizinho) in lista_tabu:
        # se estiver, vamos testar o criterio de aspiracao por objetividade
            if avaliacao(vizinho) > avaliacao(melhor_sol): # se a solucao tabu for melhor que a melhor solucao
                return vizinho # aceita essa solucao mesmo sendo tabu
        else: # se a solucao nao estiver na lista tabu, retorna ela
            return vizinho
    # se chegar ate aqui e porque nenhum vizinho foi selecionado
    print('Erro: nenhum vizinho foi selecionado como melhor solucao')

# funcao que executa o algoritmo da busca tabu
def busca_tabu(bits, BTmax, T): # tamanho da solucao
                        # nº max de iteracoes sem melhora na melhor solucao
                        # quantidade max de solucoes tabu armazenadas
    lista_tabu= [] # lista inicialmente vazia
    Iter, melhor_iter= 0, 0 # contador de iteracoes e indicador de qual iteracao teve a melhor solucao
    sol= solucao_inicial(bits)
    melhor_sol= sol[:] # inicialmente, a melhor solucao e a inicial

    print('\nSolucao inicial: ', sol)
    print('Avaliacao da solucao inicial: ', avaliacao(sol))

    while((Iter-melhor_iter) <= BTmax): # enquanto for menor ou igual a BTmax
        avaliacao_sol= avaliacao(sol) # avaliacao da solucao corrente
        vizinhos= gerar_vizinhos(sol) # gerar os vizinhos dessa solucao
        # agora vamos obter a melhor solucao dentre esses vizinhos
        melhor_vizinho= obter_melhor_vizinho(vizinhos, lista_tabu, melhor_sol, sol)
        # o melhor vizinho passa a ser a solucao corrente
```

```
        sol= melhor_vizinho[:]
        # vamos obter o movimento tabu do melhor vizinho
        pos_tabu= movimento_tabu(sol, melhor_vizinho)

        # testaremos se o tamanho da lista tabu e menor que a cardinalidade 'T'
        if len(lista_tabu) < T: # se o tamanho da lista for menor que 'T'
            lista_tabu.append(pos_tabu) # insere o movimento tabu na lista
        else:
            lista_tabu.pop(0) # retira o elemento mais velho da lista, que esta na posicao '0'
            lista_tabu.append(pos_tabu) # e insere o recem descoberto

        # verificar se o melhor vizinho e melhor que a melhor solucao
        if avaliacao(melhor_vizinho) > avaliacao(melhor_sol):
            melhor_sol= melhor_vizinho[:] # atualizamos a melhor solucao
            melhor_iter += 1 # e atualiza tambem a melhor iteracao

        Iter += 1 # e por fim, incrementa o contador de iteracoes

    print('\nTotal de iteracoes: ', Iter)
    print('Melhor iteracao: ', melhor_iter)
    return melhor_sol # retorna a melhor solucao

# testar o programa
if __name__ == '__main__':
    melhor_solucao= busca_tabu(bits= 100, BTmax= 2, T= 2)
    print('Melhor solucao: ', melhor_solucao)
    print('Avaliacao da melhor solucao: ', avaliacao(melhor_solucao))
```