Computação Bioinspirada

Projeto 1 - 18/09/2023

Lucas Cerutti Sergio - 11611BSI252
Rick Gomes Ricarte - 11921BSI200

Implementação de um algoritmo genético para o problema da mochila

1- Implementação do algoritmo genético

O algoritmo foi codificado utilizando a linguagem de programação Python, seguindo padrões de design e aplicando os conceitos vistos em sala de aula sobre o funcionamento de um algoritmo genético.

O código foi armazenado e compartilhado no GitHub, utilizando o sistema de versionamento Git, e contém tanto o código do algoritmo genético quanto o criado para a construção do relatório, além dos arquivos de entrada e saída (com as soluções)

Optamos por utilizar algumas constantes para o algoritmo, tanto para a criação da população quanto para a mutação. Após vários experimentos, os valores que se mostraram satisfatórios para a maioria das entradas foram os seguintes:

```
169  def main():
170      population_size = 100
171      generation_qtd = 10000
172      mutation_rate = 0.5
173      security_rate = 1     # 0 - 100%
```

Logo no início da execução os arquivos de input são abertos e transformados em dados para utilização no programa, onde serão executados em 5 iterações cada

```
174  for file_number in range(1,17):
175      header_line = f"Iteracao;Valor;Peso;Capacidade restante;Execucao(s)\n"
176      with open(f"output/ga/ga_{file_number}.out", "w") as output_file:
177          output_file.write(header_line)
178
179      start_time_file = time.time()
180
181      for iterator in range(1, 6):
182          start_time_iterator = time.time()
183
184          input_file_path = f"input/input{file_number}.in"
185
186          vet_value, vet_weight, capacity = get_input_values(input_file_path)
```

```python
def get_input_values(file_path):
    with open(file_path, "r") as file:
        lines = file.readlines()

    capacity = int(lines[-1].strip())

    vet_value, vet_weight = [], []
    for line in lines[1:-1]:
        numbers = re.findall(r"[0-9]+", line)
        vet_value.append(int(numbers[1]))
        vet_weight.append(int(numbers[2]))

    return vet_value, vet_weight, capacity
```

O método de seleção utilizado foi o da roleta, utilizando como peso o valor fitness dos itens (nessa implementação o fitness é simplesmente o valor do item)

```python
def select_parents(population, vet_value, vet_weight, capacity):
    vet_parents = []
    vet_fitness = []

    for chromossome in population:
        fitness_val = get_fitness_value(chromossome, vet_value, vet_weight, capacity)
        vet_fitness.append(fitness_val)

    vet_parents.append(random.choices(population, weights=vet_fitness, k=1)[0])
    vet_parents.append(random.choices(population, weights=vet_fitness, k=1)[0])

    return vet_parents, vet_fitness
```

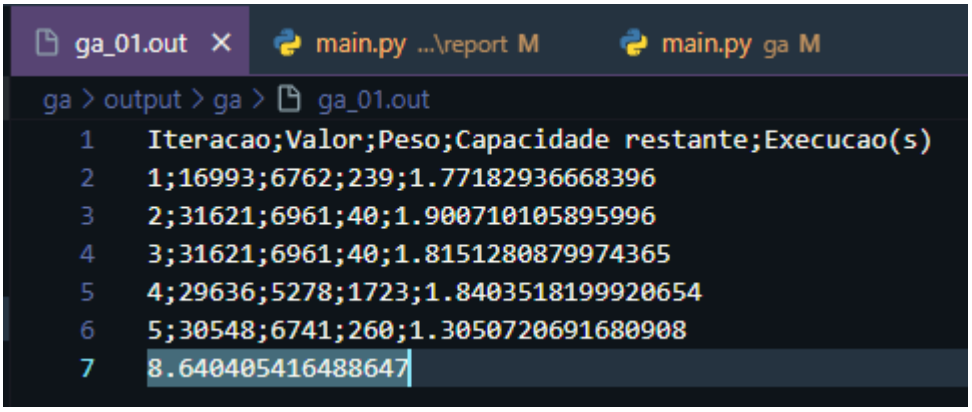
```python
def get_fitness_value(chromossome, vet_value, vet_weight, capacity):
    total_value = 0
    total_weight = 0

    for i in range(len(chromossome)):
        if chromossome[i] == 1:
            total_value += vet_value[i]
            total_weight += vet_weight[i]

    if total_weight > capacity:
        return 0.0001
    else:
        return total_value
```

Após, é realizada a mutação com dois pontos de corte e a substituição dos indivíduos dentro da população se for necessário, após comparação com os pais

```python
def crossover(vet_parents, itens_qtd):
    vet_children = []

    if itens_qtd <= 3:
        cross_point = random.randint(1, itens_qtd - 1)

        vet_children.append(vet_parents[0][:cross_point] + vet_parents[1][cross_point:])
        vet_children.append(vet_parents[1][:cross_point] + vet_parents[0][cross_point:])
    else:
        cross_point_1 = random.randint(1, itens_qtd - (itens_qtd/2))
        cross_point_2 = random.randint(itens_qtd/2, itens_qtd-1)

        vet_children.append(vet_parents[0][:cross_point_1] + vet_parents[1][cross_point_1:cross_point_
        vet_children.append(vet_parents[1][:cross_point_1] + vet_parents[0][cross_point_1:cross_point_

    return vet_children
```

```python
    vet_children = crossover(vet_parents, itens_qtd)

    random_number_mutation = random.uniform(0,1)

    if random_number_mutation <= mutation_rate:
        vet_children[0] = mutation(vet_children[0], itens_qtd)
        vet_children[1] = mutation(vet_children[1], itens_qtd)

    vet_fitness_children = []
    vet_fitness_parents = []
    population_parents_indexes = []

    population_parents_indexes.append(population.index(vet_parents[0]))
    population_parents_indexes.append(population.index(vet_parents[1]))

    vet_fitness_children.append(get_fitness_value(vet_children[0], vet_value, vet_weight, capacity))
    vet_fitness_children.append(get_fitness_value(vet_children[1], vet_value, vet_weight, capacity))

    vet_fitness_parents.append(vet_fitness[population_parents_indexes[0]])
    vet_fitness_parents.append(vet_fitness[population_parents_indexes[1]])

    vet_winners, vet_winners_fitness = compare_parents_children(vet_children, vet_fitness_children, vet_pare

    population[population_parents_indexes[0]] = vet_winners[0]
    population[population_parents_indexes[1]] = vet_winners[1]

    vet_fitness[population_parents_indexes[0]] = vet_winners_fitness[0]
    vet_fitness[population_parents_indexes[1]] = vet_winners_fitness[1]
```

Por fim, é selecionado o melhor indivíduo da população e a mesma é salva no arquivo de saída, juntamente com seu valor, peso e tempo de execução.



2- Comparação com GRASP - Algoritmo

A implementação que escolhemos para comparar com a nossa foi a que utilizou o algoritmo GRASP para a resolução do problema da mochila.

Antes de tudo, alteramos também a saída do GRASP para que pudéssemos ter acesso aos dados que queríamos usar para as análises, sendo eles os mesmos dos exportados no algoritmo GA

grasp_01.out

ga > output > grasp > grasp_01.out

```
Iteracao;Valor;Peso;Capacidade restante;Execucao(s)
1;28522;6524;477;0.0020067691802978516
2;25733;5896;1105;0.001993417739868164
3;29813;6854;147;0.0019996166229248047
4;26384;6166;835;0.0019996166229248047
5;24987;4956;2045;0.0020003318786621094
0.019999980926513672
```

Já dentro do algoritmo, o que está sendo feito é apenas a abertura de todos os arquivos de soluções (de ambos algoritmos), a extração dos dados em forma de média e desvio-padrão em cima das iterações para utilização no código e a construção de gráficos e tabela.

```
    folder_path_ga = "../output/ga"
    folder_path_grasp = "../output/grasp"

    for ind, file in enumerate(os.listdir(folder_path_ga)):
        file_path = os.path.join(folder_path_ga, file)
        vet_iteration, vet_value, vet_weight, vet_execution_time, vet_remaining_capacity  = get_file_values(file_
        value_metrics, weight_metrics, execution_time_metrics, remaining_capacity_metrics = get_all_values_metric
        results_ga.append({'File': str(ind+1)+'.out', 'Value mean': value_metrics['mean'], 'Value std': value_met
                           'Weight mean': weight_metrics['mean'], 'Weight std': weight_metrics['std'],
                           'Execution time(s) mean': execution_time_metrics['mean'], 'Execution time(s) std': exe
                           'Remaining capacity mean': remaining_capacity_metrics['mean'], 'Remaining capacity std
```

```
def get_mean_and_std(vet_value, roundNumber = False):
    mean = np.mean(vet_value)
    std_value = np.std(vet_value) # std = desvio-padrao

    if roundNumber == True:
        mean = round(mean, 4)
        std_value = round(std_value, 4)

    return mean, std_value

def get_all_values_metrics(vet_value, vet_weight, vet_execution_time, vet_remaining_capacity):
    value_mean, value_std = get_mean_and_std(vet_value)
    weight_mean, weight_std = get_mean_and_std(vet_weight)
    execution_time_mean, execution_time_std = get_mean_and_std(vet_execution_time, True)
    remaining_capacity_mean, remaining_capacity_std = get_mean_and_std(vet_remaining_capacity)

    value_metrics = {'mean': value_mean, 'std': value_std}
    weight_metrics = {'mean': weight_mean, 'std': weight_std}
    execution_time_metrics = {'mean': execution_time_mean, 'std': execution_time_std}
    remaining_capacity_metrics = {'mean': remaining_capacity_mean, 'std': remaining_capacity_std}
```

```
    generate_datatable(vet_file_names, vet_value_ga, vet_weight_ga, vet_execution_time_ga,
                       vet_value_grasp, vet_weight_grasp, vet_execution_time_grasp)

    generate_value_x_execution_time_graph(vet_file_names, vet_value_ga, vet_value_std_ga, vet_value_grasp, vet_va

    generate_weight_value_graph(vet_file_names, vet_weight_ga, vet_weight_std_ga, vet_weight_grasp, vet_weight_st

    generate_value_by_weight_pie_graph(vet_file_names, vet_weight_ga, vet_weight_std_ga, vet_weight_grasp, vet_we
```

3- Comparação com GRASP - Resultados

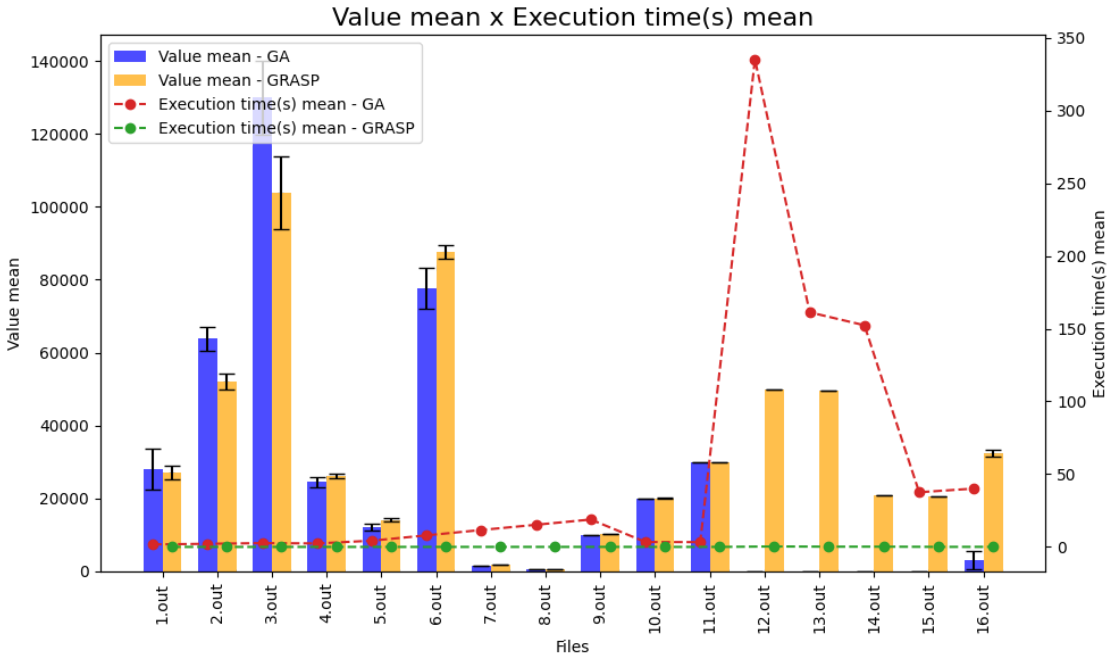
Tabela com dados gerais

General data presented as an average

	Value (GA)	Value (GRASP)	Weight (GA)	Weight (GRASP)	Time(s) (GA)	Time(s) (GRASP)
1.out	28083.8	27087.8	6540.6	6079.2	1.7266	0.002
2.out	63752.0	51941.2	9573.0	8712.8	2.1495	0.0053
3.out	129935.0	103735.4	19245.2	17796.4	2.6164	0.0031
4.out	24498.4	26145.2	19976.4	19617.8	2.4006	0.0037
5.out	12102.0	14098.0	9955.0	9628.2	4.2834	0.0056
6.out	77585.0	87491.8	49428.6	49475.8	7.9496	0.008
7.out	1446.4	1730.4	992.8	989.0	11.6393	0.0136
8.out	553.4	664.2	499.4	496.2	15.2251	0.017
9.out	9991.6	10160.6	9967.6	9984.6	18.8374	0.0215
10.out	20007.8	20063.2	19985.8	19981.2	3.389	0.0064
11.out	29973.2	29997.8	29973.2	29997.8	3.2991	0.0049
12.out	0.0	49885.0	0.0	49885.0	335.0984	0.3184
13.out	0.0	49398.0	0.0	49398.0	161.269	0.1595
14.out	0.0	20844.0	0.0	24827.6	152.3694	0.17
15.out	0.0	20676.0	0.0	1996.6	37.586	0.0285
16.out	3052.4	32441.4	1187.6	1979.4	40.0964	0.0561

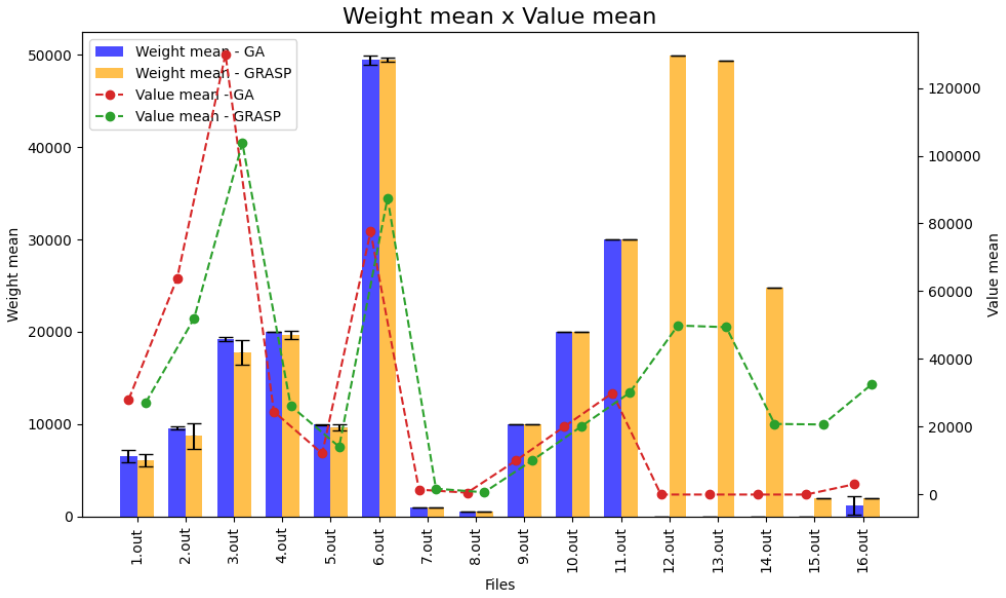
Na primeira visualização da tabela já é bem perceptível que faltam dados de valor e peso para alguns arquivos de soluções do GA. Isso porque experimentamos diversos valores de população, gerações e mutação, e os que obtiveram mais sucesso, infelizmente, não se aplicam muito bem às massas dos arquivos de input 12, 13, 14 e 15.
Além disso, é possível perceber que os algoritmos empatam na questão de soluções que aproveitaram melhor a capacidade da mochila para com o peso, porém, em questão de valor total atingido (que é o foco do problema), o algoritmo GRASP possui um desempenho muito melhor, ainda mais nos arquivos com mais itens para entrada. A respeito do tempo de execução, indiscutivelmente o GRASP se destaca, visto que mesmo no tempo mais curto do GA, o mesmo ainda é 86230% maior que o tempo do GRASP para o mesmo arquivo.

Valor médio x tempo de execução(s) médio



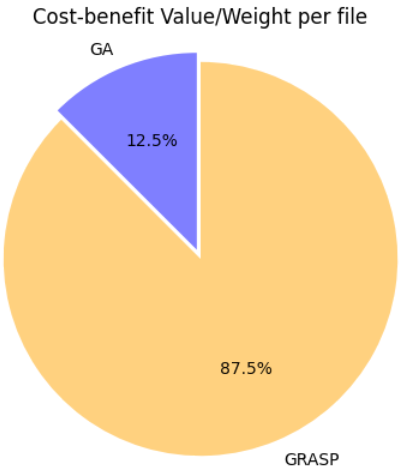
No gráfico é possível ver que nos primeiros arquivos o GA até possui valores médios mais altos que o GRASP, mas ele não possui constância no valor nem no tempo, aumentando e diminuindo em alguns momentos. Além disso, observando o desvio-padrão no topo das barras, é notável que os valores do GA diferem muito mais entre as iterações do que o GRASP, que mantém mais ou menos uma faixa parecida de valores na solução. Por fim, o tempo de execução do GRASP se mantém baixíssimo até o último arquivo.

Peso médio x valor médio



Neste gráfico é possível ver com mais clareza que, apesar de o GA possuir várias soluções com mais peso preenchido do que o GRASP, a diferença não é tão grande assim, visto que na maioria dessas soluções, o GA ainda possui valor semelhante ou muito menor que o GRASP. Além de que, no resto dos arquivos, o GA não conseguiu nem ao menos uma solução satisfatória ou, se conseguiu, foi notavelmente inferior a do GRASP.

Custo-benefício de valor/peso



Por fim, uma forma mais clara de visualizarmos a importância da relação valor/peso seria o gráfico de pizza que informa que, em 87,5% dos arquivos (14 arquivos), vale mais a pena a solução encontrada pelo GRASP, pois o valor obtido dividido pelo peso ocupado vale mais a

pena. Isso sem colocar os tempos de execução na fórmula, pois, visto os dados da tabela e do gráfico “Valor médio x tempo de execução(s) médio”, o GRASP supera em muito os resultados obtidos com a implementação GA.