Universidade Federal de Mato Grosso do Sul
Campus Ponta Porã
Teoria da Computação

Trabalho Prático I

Heurísticas para o Problema da Mochila Booleana

Aluno: Daniel de Leon Bailo da Silva
Professor: Eduardo Theodoro Bogue

Setembro
2019

Sumário

Resumo

A proposta deste trabalho consistia em desenvolver uma *Heurística Gulosa* e uma *Heurística GRASP* para o *Problema da Mochila Booleana* para as instâncias disponibilizadas para teste e comparar os resultados obtidos para as duas versões.

Porém, a fim de melhor realizar uma pesquisa mais concreta para realizar as comparações dos algoritmos, foram desenvolvidas três heurísticas gulosas e três variações de uma mesma heurística GRASP e, após comparar os resultados entre si, foi também comparado aos resultados exatos para cada instância. Logo, é possível saber exatamente a eficiência de cada heurística construída.

Este trabalho está armazenado num repositório do GitHub(https://github.com/danbailo/T1-Teoria-Computacao) e o mesmo está disponibilizado em um *Jupyter Notebook* e em um script escrito em *Python*.

1 Introdução

O problema da Mochila(knapsack problem) pode ser enunciado da seguinte forma: Dados um número $m \geq 0$, um inteiro positivo n e, para cada i em $1, ..., n$, um número $v_i \geq 0$ e um número $w_i \geq 0$, encontrar um subconjunto S de $1, ..., n$ que maximize $v(S)$ sob a restrição $w(S) \leq m$. Onde, v(S) denota a soma $\sum_{i \in S} v_i$ e, analogamente, $w(S)$ denota a soma $\sum_{i \in S} w_i$.

Os números v_i e w_i podem ser interpretados como o valor e peso respectivamente de um objeto i. O número W pode ser interpretado como a capacidade de uma mochila, ou seja, o peso máximo que a mochila comporta. O objetivo do problema é então encontrar uma coleção de objetos, a mais valiosa possível, que respeite a capacidade da mochila.

Este problema vem sendo estudado deste o trabalho de *D.G. Dantzig* [4], devido a sua utilização imediata na Indústria e na Gerencia Financeira, porém foi mais enunciado por razões teóricas, uma vez que este frequentemente ocorre pela relaxação de vários problemas de programação inteira. Toda a família de **Problemas da Mochila** requer que um subconjunto de itens sejam escolhidos, de tal forma que o somatório dos seus valores seja maximizado sem exceder a capacidade da mochila. Diferentes tipos de problemas da Mochila ocorrem dependendo da distribuição de itens e Mochilas como citado em [4]:

No problema da *Mochila 0/1(0/1 Knapsack Problem)*, cada item pode ser escolhido no máximo uma vez, enquanto que no problema da *Mochila Limitado(Bounded Knapsack Problem)* temos uma quantidade limitada para cada tipo de item. O problema da *Mochila com Múltipla Escolha(Multiple-choice Knapsack Problem)* ocorre quando os itens devem ser escolhidos de classes disjuntas, e se várias Mochilas são preenchidas simultaneamente temos o problema da *Mochila Múltiplo(Multiple Knapsack Problem)*. A forma mais geral é o problema da *Mochila com multirestrições (Multi-constrained Knapsack Problem)* o qual é basicamente um problema de *Programação Inteira Geral com Coeficientes Positivos*.

Todos os problemas da Mochila pertencem a família **NP-Hard** [4], significando que é muito improvável que possamos desenvolver algoritmos polinomiais para este problema. Porém, a despeito do tempo para o pior caso de todos os algoritmos terem tempo exponencial, diversos exemplos de grandes instâncias podem ser resolvidos de maneira ótima em fração de segundos. Estes resultados surpreendentes vem de várias décadas de pesquisa que tem exposto as propriedades estruturais especiais do Problema da Mochila, que tornam o problema tão relativamente fácil de resolver.

Neste trabalho, foram desenvolvidas três heurísticas gulosas e uma heurística GRASP que resolvem o Problema da Mochila $0/1$, que consiste em escolher n itens, tais que o somatório das utilidades é maximizado sem que o somatório dos pesos extrapolem a capacidade da Mochila. No entanto, dada as instâncias para realizar os experimentos, foram comparadas os resultados obtidos a partir das heurísticas e comparados com o resultado exato para cada instância, a fim de verificar a eficiência de cada heurística programada.

1.1 Tramento de Problemas NP-Hard

Para tratar um problema NP-Hard, devemos sacrificar uma das seguintes características:

1. Resolver o problema na otimalidade.
2. Resolver o problema em tempo polinomial.

Para isto, podemos desenvolver:

- Algoritmos Aproximados Sacrificam 1.
- Algoritmos Exatos Sacrificam 2.
- Heurísticas Sacrificam 1 e possivelmente 2.

1.2 Heurísticas

Heurísticas são processos cognitivos empregados em decisões não racionais, sendo definidas como estratégias que ignoram parte da informação com o objetivo de tornar a escolha mais fácil e rápida. [2] Heurísticas rápidas e frugais (fast and frugal heuristics) correspondem a um conjunto de heurísticas propostas por Gigerenzer e que empregam tempo, conhecimento e computação mínimos para fazer escolhas adaptativas em ambientes reais. [3]

Existem três passos cognitivos fundamentais na selecção de uma heurística:

- Procura – As decisões são tomadas entre alternativas e por esse motivo há uma necessidade de procura ativa;
- Parar de procurar – A procura por alternativas tem que terminar devido as capacidades limitantes da mente humana;
- Decisão – Assim que as alternativas estiverem encontradas e a procura for cessada, um conjunto final de heurísticas são chamadas para que a decisão possa ser tomada. [3]

1.2.1 GRASP

GRASP (greedy randomized adaptive search procedure) é uma metaheurístisca multistart para problemas de otimização combinatória, na qual cada iteração consiste basicamente de duas fases: construção e busca local. A fase de construção cria uma solução viável cuja vizinhança é investigada até que um mínimo local seja encontrado durante a fase de busca local. A melhor solução geral é mantida como resultado [1].

Como diversos métodos construtivos, a aplicação do GRASP consiste em criar uma solução inicial e depois efetuar uma busca local para melhorar a qualidade da solução. Seu diferencial para outros métodos está na geração dessa solução inicial, baseada nas três primeiras iniciais de sua sigla em inglês: gulosa (Greedy), aleatória (Randomized) e adaptativa (Adaptive).

2 Métodos

Nesta seção serão abordados os métodos que foram utilizados para obter os resultados que serão discutidos posteriormente. Todo o projeto foi escrito em *Python* e está disponível no repositório do GitHub https://www.github.com/danbailo/T1-Teoria-Computacao.

2.1 Algoritmo Exato

Este foi o algoritmo utilizado como base para os resultados ótimos; para a obtenção destes resultados, o algoritmo foi executado numa máquina presente na nuvem do Google(https://colab.research.google.com), pois assim, como o mesmo é executado de forma sequencial, o hardware desta máquina é mais potente e assim os resultados poderiam ser obtidos de forma mais rápida.

```
        import numpy as np

        def exact(number_items, weight_max, values_items, weight_items):
            K = np.zeros((number_items+1, weight_max+1), dtype=np.int32)
            for i in range(number_items+1):
                for weight in range(weight_max+1):
                    if i==0 or weight==0: K[i][weight] = 0
                    elif weight_items[i-1] <= weight: K[i][weight] =␣
↪max(values_items[i-1] + K[i-1][weight-weight_items[i-1]],␣
↪K[i-1][weight])
                    else: K[i][weight] = K[i-1][weight]
            return int(K[number_items][weight_max])
```

Algoritmo 1. Exato

2.2 Heurísticas Gulosas

Nesta primeira Heurística, temos que os pesos dos itens foram ordenados de forma crescente.

```
        def crescent(number_items, weight_max, values_items,␣
↪weight_items):
            items = {}
            for i in range(number_items): items[i] =␣
↪values_items[i],weight_items[i]
            items = sorted(items.values())
            result_final = 0
            weight = 0
            for values_items,weight_items in items:
                if weight_items+weight <= weight_max:
                    result_final += values_items
                    weight += weight_items
            return result_final
```

Algoritmo 2. Crescente

Nesta segunda Heurística, temos que os pesos dos itens foram ordenados de forma decrescente.

```
        def decrescent(number_items, weight_max, values_items,␣
↪weight_items):
            items = {}
            for i in range(number_items): items[i] =␣
↪values_items[i],weight_items[i]
            items = sorted(items.values(), reverse=True)
            result_final = 0
            weight = 0
            for values_items,weight_items in items:
                if weight_items+weight <= weight_max:
                    result_final += values_items
                    weight += weight_items
            return result_final
```

Algoritmo 3. Decrescente

Nesta terceira Heurística, temos que os os itens foram ordenados de acordo com o resultado da razão $\frac{valor}{peso}$ de forma crescente.

```
    import numpy as np

    def efficiency(number_items, weight_max, values_items,␣
↪weight_items):
        efficiency = np.divide(values_items, weight_items)
        items = {}
        for i in range(number_items): items[i] = efficiency[i],␣
↪values_items[i], weight_items[i]
        items = sorted(items.values(), reverse=True)
        result_final = 0
        weight = 0
        for _,values_items, weight_items in items:
            if weight_items+weight <= weight_max:
                result_final += values_items
                weight += weight_items
        return result_final
```

Algoritmo 4. Eficiente

2.3 Heurística GRASP

fase de construcao

[10]:
```
    import numpy as np
    import random
    random.seed(42)

    def semi_greedy_construction(window, number_items, weight_max,␣
↪values_items, weight_items):
        efficiency = np.divide(values_items, weight_items)
        items = {}
        for i in range(number_items):
            items[i] = efficiency[i], values_items[i], weight_items[i]
        items = sorted(items.values(), reverse=True)
        result_final = []
        value = 0
        weight = 0
        aux = items[:]
        while len(items) > 0 and weight < weight_max:
            if len(items) >= window: tmp_window = window
            else: tmp_window = len(items)
            index = random.randint(0,tmp_window-1)
            value_item = items[index][1]
            weight_item = items[index][2]
            if weight_item+weight <= weight_max:
                result_final.append(items.pop(index))
                value += value_item
                weight += weight_item
```

```
            else: items.pop(index)
        solution = np.zeros(number_items,dtype=np.int16)
        for aux_value in aux:
            if aux_value in result_final: solution[aux.
↪index(aux_value)] = 1
        return solution, aux, value
```

busca local

[8]:
```
    def local_search(solution, aux, value, weight_max):
        all_solutions = []
        solution_aux = solution[:]
        for i in range(len(solution)):
            if solution[i] == 1: solution[i] = 0
            elif solution[i] == 0: solution[i] = 1
            all_solutions.append(solution[:])
            solution[:] = solution_aux
        new_solution = solution_aux[:]
        for solution in all_solutions:
            new_value = 0
            new_weight = 0
            for j in range(len(solution)):
                if solution[j] == 1:
                    new_value += aux[j][1]
                    new_weight += aux[j][2]
            if new_weight <= weight_max and new_value > value:
                new_solution = solution[:]
                value = new_value
        if new_solution.all() == solution_aux.all(): return value
        return local_search(solution, aux, value, weight_max)
```

grasp

[9]:
```
    def grasp(max_it, window, number_items, weight_max, values_items,␣
↪weight_items):
        best_solution = 0
        for i in range(max_it):
            solution, aux, value = semi_greedy_construction(window,␣
↪number_items, weight_max, values_items, weight_items)
            solution = local_search(solution, aux, value, weight_max)
            if solution > best_solution:
                best_solution = solution
                verify = 0
            if solution != best_solution:
                verify += 1
                if verify == max_it*0.1: return best_solution
        return best_solution
```

3 Avaliação Experimental

Como podemos ver na figura abaixo, existe uma grande discrepância entre os resultados obtidos para algumas instâncias.

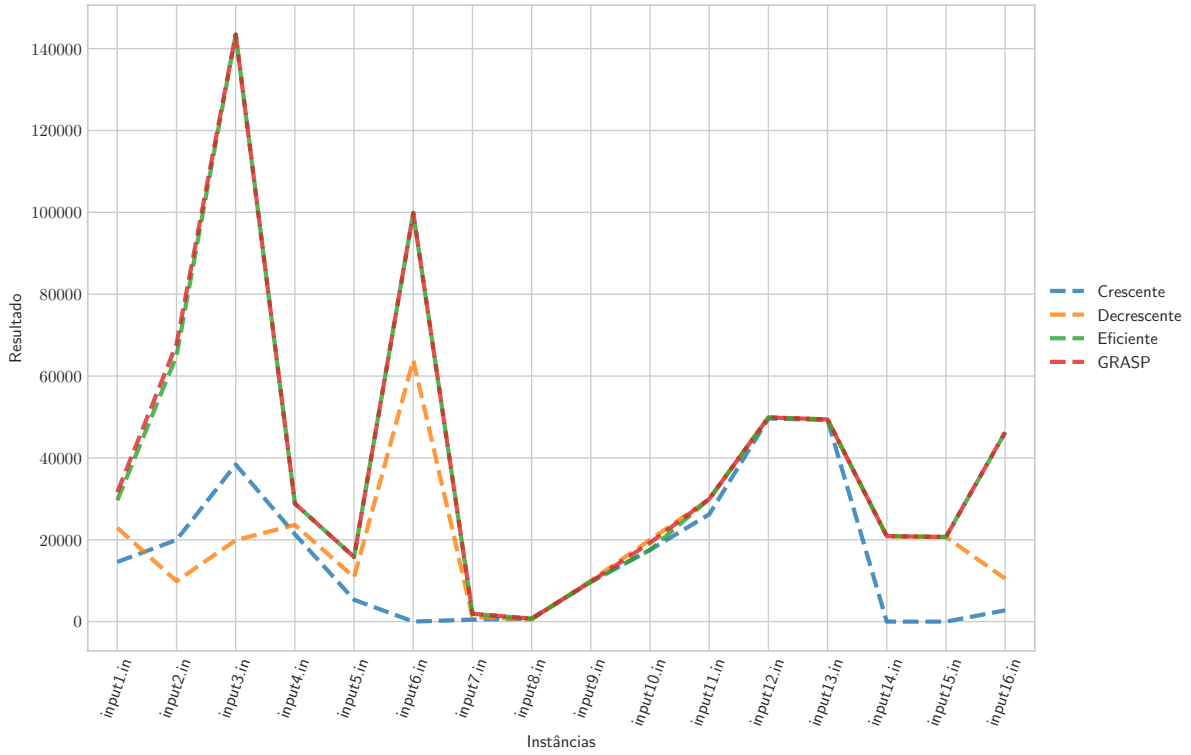

Figura 1. Comparação entre resultados das Heurísticas Gulosas.

Logo, a seguinte questão é levantada: "Dentre as Heurísticas Gulosas programadas, qual teria melhor eficiência quando comparado ao resultado de um algoritmo ótimo?"

Como podemos ver na figura abaixo, aparentemente, a Heurística que mais se comparada ao resultado ótimo, é a "Eficiente".

A fim poder afirmar com mais clareza o resultado do gráfico acima, foi realizado o cálculo da correlação dos valores relacionado ao resultado exato.

Pode se ver que, a correlação *linha/coluna* dos resultados da Heurística "Eficiente" quando comparada ao "Exato", é de quase 100%, quanto as outras, não chegam a 50%.

4 Interpretação dos Resultados

A correlacao nao consiste na porcentagem de acertar oou nao um resultado, e sim o quao proximo foi este resultado do resultado exato

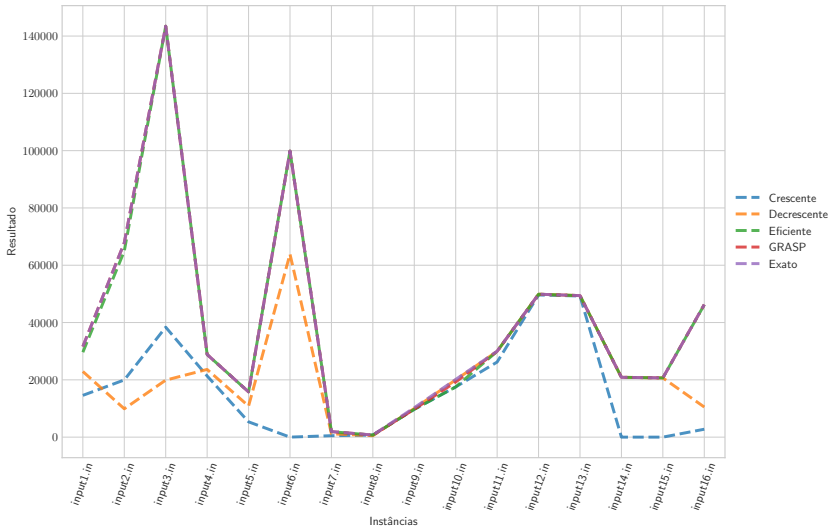

Figura 2. Comparação entre resultados das Heurísticas Gulosas.

5 Conclusão

Referências

[1] Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[2] Gerd Gigerenzer and Wolfgang Gaissmaier. Heuristic decision making. *Annual Review of Psychology*, 62(1):451–482, 2011. PMID: 21126183.

[3] Gerd Gigerenzer and Peter M Todd. *Simple heuristics that make us smart*. Oxford University Press, USA, 1999.

[4] David Pisinger. Algorithms for knapsack problems. 1995.

Figura 3. Heatmap para ver a correlação dos resultados obtidos.