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

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* [3], 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 [3]:

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** [3], 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-Completo, 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. [1] 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. [2]

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. [2]

2 Métodos

```
def crescent(number_items, weight_max, values_items, weight_items):
    items = {}
    for it in range(number_items):
        items[it] = values_items[it],weight_items[it]
    del values_items, weight_items

    items = sorted(items.values())
    result_final = []
    weight = []

    for values_items,weight_items in items:
        if weight_items+sum(weight) < weight_max and weight_items <␣
↪weight_max:
            result_final.append(values_items)
            weight.append(weight_items)
    return sum(result_final)
```

```
def decrescent(number_items, weight_max, values_items, weight_items):
    items = {}
    for it in range(number_items):
        items[it] = values_items[it],weight_items[it]
    del values_items, weight_items

    items = sorted(items.values(), reverse=True)
    result_final = []
    weight = []

    for values_items,weight_items in items:
        if weight_items+sum(weight) < weight_max and weight_items <␣
↪weight_max:
            result_final.append(values_items)
            weight.append(weight_items)
    return sum(result_final)
```

```
import numpy as np

def efficiency(number_items, weight_max, values_items, weight_items):
    value_efficiency = np.array(values_items)
    weight_items_efficiency = np.array(weight_items)
    efficiency = value_efficiency/weight_items_efficiency
    efficiency = list(efficiency)

    items = {}
    for i in range(number_items):
        items[i] = efficiency[i], values_items[i], weight_items[i]
```

```
    del values_items, weight_items
    del value_efficiency, weight_items_efficiency

    items = sorted(items.values(), reverse=True)

    result_final = []
    weight = []

    for _,values_items, weight_items in items:
        if weight_items+sum(weight) < weight_max and
        weight_items < weight_max:
            result_final.append(values_items)
            weight.append( weight_items)
    return sum(result_final)
```

```
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][w] = 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 K[number_items][weight_max]
```

3 Análise dos Resultados

4 Interpretação dos Resultados

5 Conclusão

Referências

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

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

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