Universidade Federal de Lavras

Problema da Mochila

Complexidade e Projeto de Algoritmos (GCC253)

Integrantes:
Davi Hermógenes Siqueira
André Marçal Medeiros
Lucas Gomes Colombo
Francisco Afonso Neto
Eduardo Dezena Golçalves

Introdução

O Problema da Mochila (Knapsack) é um clássico da teoria da computação e otimização combinatória. Ele é usado para demonstrar a dificuldade de problemas de otimização e tem aplicações em diversas áreas, como economia, logística e até mesmo planejamento de recursos.

Definição

Imagine que você é um aventureiro que precisa escolher itens para colocar em sua mochila. Cada item tem um peso e um valor. A mochila tem uma capacidade limitada, o que significa que não pode carregar mais do que um certo peso. O objetivo do problema da mochila é determinar quais itens você deve escolher para maximizar o valor total, sem exceder a capacidade de peso da mochila.

Formalização do Problema

Dados de entrada:

- Um conjunto de n itens, onde cada item tem:
  - Peso wi
  - Valor vi
- Uma capacidade máxima W para a mochila.

Objetivo:

- Escolher um subconjunto de itens cujo peso total Σwi seja menor ou igual a W, e cujo valor total Σvi seja maximizado.

Exemplo Ilustrativo

- Itens disponíveis:

1. Item A: Peso = 2, Valor = 3
2. Item B: Peso = 3, Valor = 4
3. Item C: Peso = 4, Valor = 5
4. Item D: Peso = 5, Valor = 8

- Capacidade da Mochila: W=8

- Soluções:

1. Item A+Item B = Peso 5, Valor 7
2. Item A+Item C = Peso 6, Valor 8
3. Item B + Item D = Peso 8, Valor 12 (Melhor opção)
4. Item C + Item D = Peso 9, Valor 13 (Não cabe na mochila)
5. etc

Universidade Federal de Lavras

Força-Bruta

- Função calcularSolucao:
  - Essa Função chama a função recursiva gerarCombinacoes para gerar todas as possíveis combinações de itens.
  - Após a execução, imprime o melhor benefício e a solução correspondente.
- Função gerarCombinacoes:
  - Critérios de Parada: Se o índice atual (indice) alcançar o tamanho da lista de itens ou se o peso atual (pesoAtual) exceder a capacidade da mochila, verifica-se se a solução atual é melhor que a armazenada e, se for, atualiza a melhor solução.
- Recursão: São feitas duas chamadas recursivas:
  - A primeira inclui o item atual na solução e avança para o próximo índice.
  - A segunda exclui o item atual da solução e também avança para o próximo índice.

```
ProblemaMochilaForcaBruta(capacidadeMochila, pesoItens, beneficioItens)
    melhorBeneficio ← 0
    melhorSolucao ← lista vazia

    calcularSolucao()
        gerarCombinacoes(0, lista vazia, 0, 0)
        PRINT "Benefício da solução ótima: " + melhorBeneficio
        PRINT "Itens da solução ótima: " + melhorSolucao

    gerarCombinacoes(indice, itensAtuais, pesoAtual, beneficioAtual)
        IF indice = SIZE(pesoItens) OR pesoAtual > capacidadeMochila
            IF beneficioAtual > melhorBeneficio AND pesoAtual ≤ capacidadeMochila
                melhorBeneficio ← beneficioAtual
                melhorSolucao ← COPY(itensAtuais)
            RETURN

        ADD pesoItens[indice] TO itensAtuais
        gerarCombinacoes(indice + 1, itensAtuais, pesoAtual + pesoItens[indice],
        beneficioAtual + beneficioItens[indice])

        REMOVE pesoItens[indice] FROM itensAtuais
        gerarCombinacoes(indice + 1, itensAtuais, pesoAtual, beneficioAtual)
```

Problema da Mochila

Universidade Federal de Lavras

Força-Bruta

- Número de Subconjuntos:
  - O algoritmo de força bruta explora todas as possíveis combinações de itens para encontrar a solução ótima. Como cada item pode estar presente ou ausente na solução, o número total de subconjuntos é 2^n
- Geração de Combinações:
  - Para cada subconjunto, o algoritmo calcula o peso e o benefício correspondentes e, portanto, avalia 2^n combinações.
- Portanto, a complexidade total do algoritmo é O(2^n), que faz com que ele seja impraticável para grandes valores de ‘n’

```
ProblemaMochilaForcaBruta(capacidadeMochila, pesoItens, beneficioItens)
    melhorBeneficio ← 0
    melhorSolucao ← lista vazia

    calcularSolucao()
        gerarCombinacoes(0, lista vazia, 0, 0)
        PRINT "Benefício da solução ótima: " + melhorBeneficio
        PRINT "Itens da solução ótima: " + melhorSolucao

    gerarCombinacoes(indice, itensAtuais, pesoAtual, beneficioAtual)
        IF indice = SIZE(pesoItens) OR pesoAtual > capacidadeMochila
            IF beneficioAtual > melhorBeneficio AND pesoAtual ≤ capacidadeMochila
                melhorBeneficio ← beneficioAtual
                melhorSolucao ← COPY(itensAtuais)
            RETURN

        ADD pesoItens[indice] TO itensAtuais
        gerarCombinacoes(indice + 1, itensAtuais, pesoAtual + pesoItens[indice],
        beneficioAtual + beneficioItens[indice])

        REMOVE pesoItens[indice] FROM itensAtuais
        gerarCombinacoes(indice + 1, itensAtuais, pesoAtual, beneficioAtual)
```

Problema da Mochila

Heurística

- Abordagem gulosa;
- **Vantagens:** Rápida, fácil de implementar, boa para problemas grandes;
- **Limitações:** Não garante a solução ótima; pode falhar em casos específicos.

Universidade Federal de Lavras

Pseudocódigo

```
INICIAR ProblemaMochilaHeuristica(capacidadeMochila, pesoItens, beneficioItens)
        // Inicializar a fila de prioridade
        razaoPesoBeneficio ← FilaDePrioridade(vazia,
    comparadorPorRazaoBeneficioPesoDecrescente)

        PROCEDIMENTO calcularSolucao()
        // Definir prioridades baseado na razão benefício/peso
        CHAMAR definirPrioridades()

        capacidadeAtual ← 0
        elementosSolucao ← lista vazia
```

Problema da Mochila

Universidade Federal de Lavras

Pseudocódigo

```
    // Enquanto houver itens na fila e a capacidade da mochila não for excedida
    ENQUANTO razaoPesoBeneficio não estiver vazia E capacidadeAtual ≤
capacidadeMochila FAZER
        elemento ← remover(razaoPesoBeneficio)

        SE pesoItens[elemento.indice] + capacidadeAtual ≤
capacidadeMochila ENTÃO
            // Adicionar o item à solução e atualizar a capacidade atual
            capacidadeAtual ← capacidadeAtual +
pesoItens[elemento.indice]
            ADICIONAR elemento.indice A elementosSolucao
        SENÃO
            // Interromper o loop se o próximo item não couber na mochila
            PARAR
        FIM SE
    FIM ENQUANTO
```

Problema da Mochila

Universidade Federal de Lavras

Pseudocódigo

```
    IMPRIMIR elementosSolucao
    FIM PROCEDIMENTO

    PROCEDIMENTO definirPrioridades()
    PARA i ← 0 ATÉ TAMANHO(pesoItens) - 1 FAZER
        razao ← beneficioItens[i] / pesoItens[i]
        ADICIONAR (razao, i) A razaoPesoBeneficio
    FIM PARA
    FIM PROCEDIMENTO
FIM INICIAR
```

Problema da Mochila

Análise Experimental

Objetivo

- O objetivo é comparar o desempenho de diferentes algoritmos para o problema da mochila, e entender como esses algoritmos se comportam em termos de tempo de execução.
- Além é claro de analisar a complexidade de cada um e relacionar com os tempos medidos

Heurística x Força Bruta (Relembrando)

Heurística

Como sabemos, uma heurística não garante solução ótima para o problema, embora em alguns casos ela possa chegar a solução ótima, o objetivo dela não é esse, é simplesmente responder em um tempo satisfatório e trazer uma solução boa.

Força Bruta

O algoritmo força bruta tenta todas as combinações até encontrar a solução ótima, ele é direto e simples, porém muito ineficiênte para problemas grandes.

Universidade Federal de Lavras

Complexidade

Força Bruta O (2^N)

- a complexidade do força bruta se resume a quantidade n de itens a serem considerados, pois ele funciona de modo a explorar todos os subconjuntos a fim de obter o conjunto com melhor relação custo/benefício sem ultrapassar o limite da mochila.

Heurística O (N LOG N)

- a complexidade da heurística é ditada pela ordenação, ordena os elementos em função do custo-benefício, e coloca os melhores na frente. No nosso caso a heap é uma max-priority-queue, cada inserção custa Log N, e é realizada N vezes.

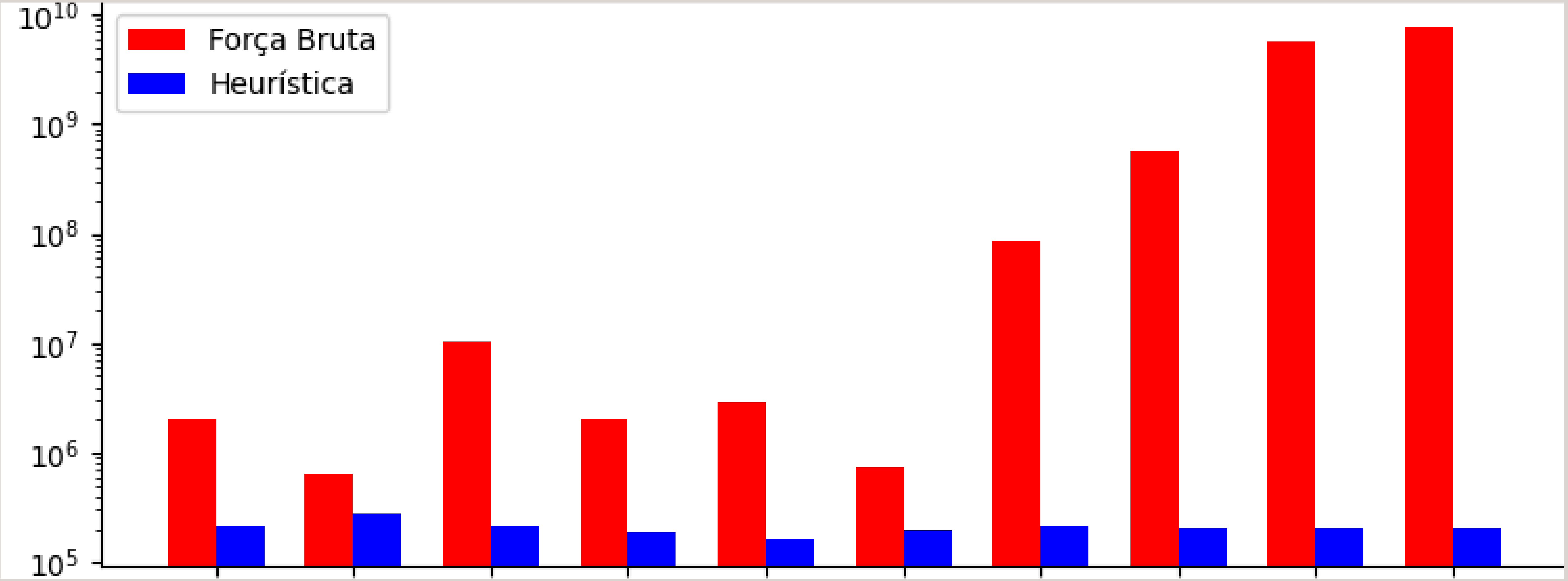
Testes

Sobre os testes

- Para os testes utilizamos entradas de tamanho crescente que variam de 1 até 30 itens. Com capacidades variáveis.
- Cada entrada foi executada 10 vezes e ao fim obtemos a média aritmética e plotamos em um gráfico utilizando python (bibliotecas matplotlib e numpy).

Universidade Federal de Lavras

Resultados e Discussões



Fonte: Autoria Própria

Problema da Mochila

Resultados e Discussões

Questão

- E se compararmos ambas as execuções com um terceiro algoritmo, porém de programação dinâmica ?

Resultados e Discussões

Programação Dinâmica (Bottom-Up)

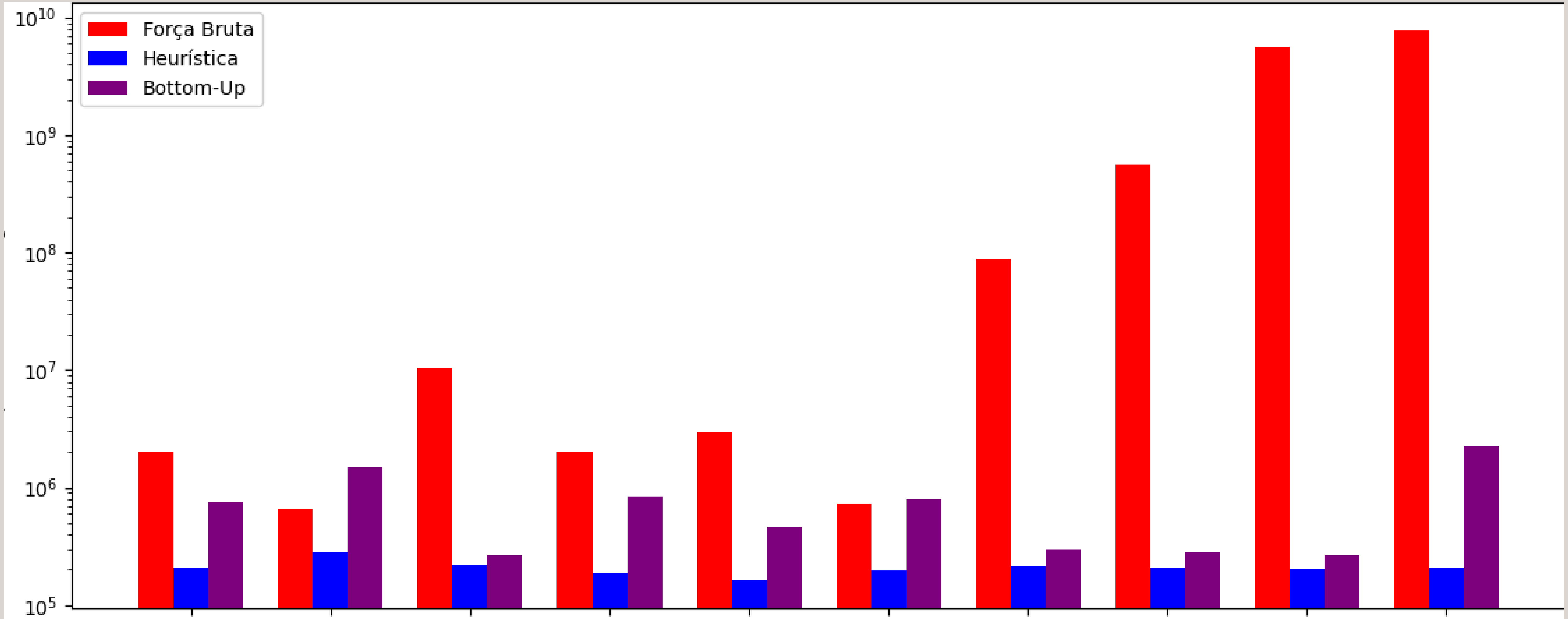
A abordagem usando prog. dinâmica utiliza ‘memoization’ e uma ordem ‘esperta’ de resolução de subproblemas. Com ‘esperta’ nos referimos a resolver os subproblemas menores primeiro, isto é, todos os subproblemas de tamanho 0, depois 1, 2,..., N.

Complexidade

O algoritmo bottom-up preenche uma tabela bidimensional de tamanho ((n+1) x (W+1)). Cada entrada da tabela é preenchida em tempo constante.

Universidade Federal de Lavras

Resultados e Discussões



Fonte: Autoria Própria

Problema da Mochila

Universidade Federal de Lavras

Assintoticamente Acurados?

Força Bruta: com sua complexidade exponencial, faz-se impraticável utilizar este algoritmo para entradas de problemas muito grandes. Imagine N=40 itens para calcular os subconjuntos, seria 2^40 = 1.0995116e+12

Heurística: Embora não garanta a solução ótima, tem tempo de execução muito baixo, geralmente linear ou polinomial (polinomial no nosso caso), como observado nos seus resultados. Contudo, a solução pode ser subótima.

Programação Dinâmica (Bottom-Up): tem complexidade Polinomial O(N*W). Os tempos de execução refletem sua eficiência, principalmente em instancias maiores. mas pode ter limitações de memória para problemas grandes.

Problema da Mochila

Corretude e Considerações Finais

Artigo (UFAM): Abordagens para resolver o problema da mochila

Resultados e implementações foram bastante similares, porém o autor testou três estratégias gulosas, relação valor/peso, selecionar itens de maior valor, e itens de menor peso.

Referências

SOUZA, Éfren Lopes de; RAFAEL, Erik Alexander Landim. Abordagens para resolver o problema da mochila 0/1. Revista IGAPÓ, 2009. Disponível em: Colocar aqui que pesquisamos outras implementações para ter uma noção da direção.. Acesso em: 23 ago. 2024.

DONG, Jian; WANG, Zhiyu; MO, Jinjun. A Phase Angle-Modulated Bat Algorithm with Application to Antenna Topology Optimization. Applied Sciences, v. 11, n. 5, p. 2243, 2021. Disponível em: Your paragraph text. Acesso em: 23 ago. 2024.

Obrigado