INSTITUTO FEDERAL DO ESPÍRITO SANTO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA – PPCOMP

BRUNA RUPP RUELA

TRABALHO 3 - NP-COMPLETO COM CLOJURE

Serra
2023

BRUNA RUPP RUELA

TRABALHO 3 - NP-COMPLETO COM CLOJURE

Trabalho apresentado ao Programa de Pós-Graduação em Computação Aplicada – PPCOMP do Instituto Federal do Espírito Santo, como requisito parcial de obtenção de nota para aprovação da Disciplina de Teoria da Computação.

Professor: Prof. Dr. Jefferson O. Andrade

Serra
2023

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Os problemas NP-completos são uma classe de problemas computacionais que são muito difíceis de resolver. Eles são tão difíceis que, se encontrarmos uma maneira de resolver um problema NP-completo de forma eficiente, também podemos resolver todos os outros problemas NP-completos de forma eficiente.

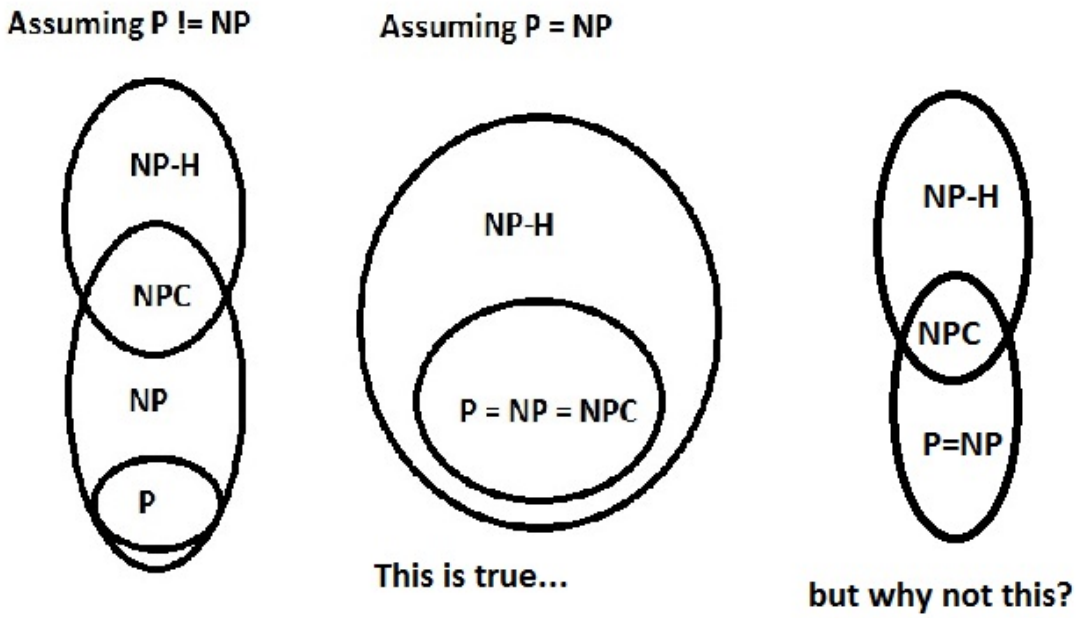
O Clojure é uma linguagem de programação funcional e é definido em termos de avaliação de estruturas de dados e não em termos de sintaxe de fluxos/arquivos de caracteres (CLOJURE, 2008-2022). É frequentemente usada para problemas de inteligência artificial e aprendizado de máquina. É uma linguagem relativamente nova, mas tem sido usada para resolver uma variedade de problemas NP-completos, incluindo:

O problema do caixeiro viajante: encontrar o caminho mais curto para visitar uma série de cidades. O problema do empacotamento: encontrar a maneira mais eficiente de empacotar objetos em uma caixa. O problema da programação linear: encontrar a solução ótima para um problema de otimização linear.

1.1 PROBLEMAS NP-COMPLETO

Um problema NP-completo é um problema que é difícil de resolver, mas fácil de verificar. Isso significa que é fácil dizer se uma solução para um problema NP-completo é correta, mas é difícil encontrar uma solução correta. A Figura 2 ilustra uma dúvida de um usuário do site de resolução de problemas *Stackoverflow*, a qual faz o questionamento se é possível termos problemas de classe P igual a NP que por sua vez é também igual a NP-Completo e quais seriam as implicações se existesse essa prova, esse exemplo reforça o quão essa classe de problemas é difícil de resolver e como promove muitos desafios de soluções entre os pesquisadores, programadores e interessados em soluções de problemas.

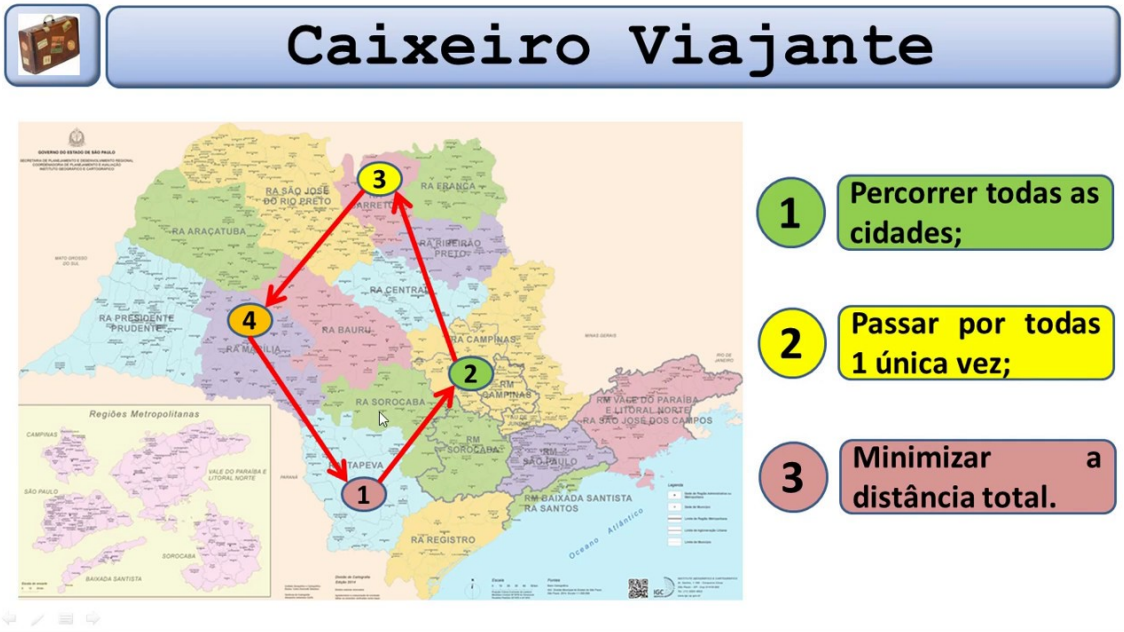
Figura 1 – Classes de problemas P, NP e NP Completo



Fonte: stackoverflow (2014).

Um exemplo de problema NP-completo é o problema do caixeiro viajante. Este problema consiste em encontrar o caminho mais curto para visitar uma série de cidades. É um problema difícil porque há muitas possíveis soluções para o problema, e cada solução deve ser testada para ver se é a melhor (CUNHA; BONASSER; ABRAHÃO, 2002).

Figura 2 – O problema do caixeiro Viajante



Fonte: AnibalAzevedo (2021).

1.1.1 Soluções para problemas NP-completos

Existem dois tipos principais de soluções para problemas NP-completos:

Soluções exatas: Essas soluções encontram a solução ótima para o problema. Soluções aproximadas: Essas soluções encontram uma solução que é boa, mas não necessariamente ótima. Soluções exatas para problemas NP-completos são geralmente muito difíceis de encontrar. No entanto, soluções aproximadas podem ser encontradas usando uma variedade de técnicas, incluindo:

Programação linear: Essa técnica usa equações lineares para representar o problema. Busca local: Essa técnica começa com uma solução inicial e então faz pequenas alterações para melhorar a solução. Algoritmos genéticos: Esses algoritmos simulam a evolução natural para encontrar soluções para problemas.

1.1.2 O Clojure para problemas NP-completos

O Clojure é uma linguagem bem-adaptada para problemas NP-completos porque é uma linguagem funcional. As linguagens funcionais são boas para problemas que envolvem a manipulação de dados recursivamente. O Clojure também oferece uma variedade de bibliotecas e ferramentas que podem ser usadas para resolver problemas NP-completos. Essas bibliotecas incluem:

clojure.math.combinatorics: Essa biblioteca fornece funções para trabalhar com combinações e permutações. clojure.set: Essa biblioteca fornece funções para trabalhar com conjuntos. clojure.core.logic: Essa biblioteca fornece funções para trabalhar com lógica.

1.2 OBJETIVO DO PROJETO

Desenvolver um programa em Clojure que implemente algoritmos para resolver um problema NP-Completo específico. O projeto visa fornecer uma compreensão prática dos desafios associados à resolução de problemas NP-Completos e explorar soluções heurísticas ou exatas.

1.3 O PROBLEMA DA MOCHILA

O Problema da Mochila, ou *"Knapsack Problem"* em inglês, é um problema de otimização combinatória que se enquadra na classe de problemas NP-completos. Este problema é clássico na teoria da computação e na pesquisa operacional. A formulação básica do problema é a seguinte:

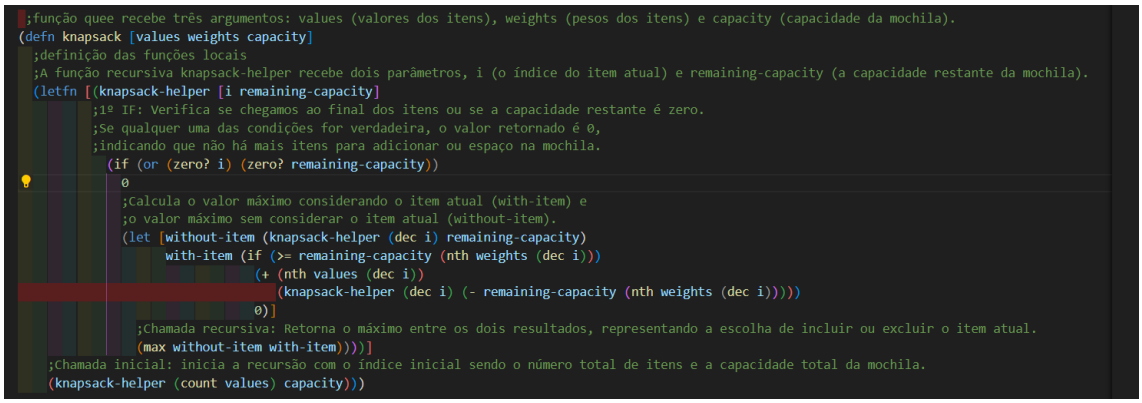
Dada uma mochila de capacidade limitada e um conjunto de itens, cada um com um peso e um valor associado, o objetivo é determinar a combinação de itens a ser colocada na

mochila de forma a maximizar o valor total, respeitando a capacidade da mochila. O Problema da Mochila é considerado NP-completo porque não se conhece um algoritmo eficiente (polinomial) para resolvê-lo em todos os casos. Em outras palavras, não há um algoritmo conhecido que possa encontrar a solução ótima em tempo polinomial em relação ao tamanho da entrada do problema (CORMEN et al., 2009).

1.3.1 Implementação de Algoritmos Exatos para o problema da Mochila

O código da Figura 3 usa uma abordagem recursiva para o problema da Mochila. Essa implementação é mais fácil de entender, mas é menos eficiente do que a abordagem de Programação Dinâmica para instâncias grandes do problema.

Figura 3 – Problema da Mochila usando Clojure



Fonte: Criado pela autora.

O codigo completo está no arquivo tabalho3.zip.

1.3.2 Implementação de Algoritmos Heurísticos para o problema da Mochila

Um algoritmo heurístico é um método de solução de problemas ou tomada de decisões que utiliza abordagens aproximadas e estratégias práticas, em vez de garantir uma solução ótima. Essas técnicas são frequentemente aplicadas em situações em que encontrar uma solução exata é computacionalmente inviável ou muito demorado (CORMEN et al., 2012). Um algoritmo heurístico para o Problema da Mochila é o algoritmo guloso (greedy). O algoritmo guloso faz escolhas locais ótimas em cada etapa, na esperança de que essas escolhas levem a uma solução globalmente ótima. No caso do Problema da Mochila, o algoritmo guloso pode ser implementado ordenando os itens de acordo com uma métrica específica (por exemplo, valor/peso) e, em seguida, selecionando os itens na ordem ordenada até que a capacidade da mochila seja atingida .

A Figura 4 mostra um trecho do problema onde o código primeiro calcula a razão valor/peso para cada item e, em seguida, ordena os itens com base nessa razão em ordem decrescente. Em seguida, percorre a lista ordenada de itens, selecionando cada item até que a capacidade

da mochila seja atingida. O resultado é impresso no console, mostrando o valor máximo que pode ser obtido e os itens selecionados.

Figura 4 – Algoritmo guloso aplicado ao problema da mochila

```
;função quee recebe três argumentos: values (valores dos itens), weights (pesos dos itens) e capacity (capacidade da mochila).
(defn knapsack [values weights capacity]
  ;definição das funções locais
  ;A função recursiva knapsack-helper recebe dois parâmetros, i (o índice do item atual) e remaining-capacity (a capacidade restante da mochila).
  (letfn [(knapsack-helper [i remaining-capacity]
            ;1º IF: Verifica se chegamos ao final dos itens ou se a capacidade restante é zero.
            ;Se qualquer uma das condições for verdadeira, o valor retornado é 0,
            ;indicando que não há mais itens para adicionar ou espaço na mochila.
            (if (or (zero? i) (zero? remaining-capacity))
              0
              ;Calcula o valor máximo considerando o item atual (with-item) e
              ;o valor máximo sem considerar o item atual (without-item).
              (let [without-item (knapsack-helper (dec i) remaining-capacity)
                    with-item (if (>= remaining-capacity (nth weights (dec i)))
                                (+ (nth values (dec i))
                                   (knapsack-helper (dec i) (- remaining-capacity (nth weights (dec i)))))
                                0)]
                ;Chamada recursiva: Retorna o máximo entre os dois resultados, representando a escolha de incluir ou excluir o item atual.
                (max without-item with-item))))]
    ;Chamada inicial: inicia a recursão com o índice inicial sendo o número total de itens e a capacidade total da mochila.
    (knapsack-helper (count values) capacity)))
```

Fonte: Criado pela autora.

1.4 Resultados

comparando as duas estratégias utilizadas para resolver o Problema da Mochila: a abordagem recursiva e a heurística gulosa temos que,

Abordagem Recursiva

Complexidade de Tempo:

Exponencial: $O(2^n)$, onde n é o número de itens. Cada chamada recursiva gera duas novas chamadas, levando a uma explosão combinatorial.

Exatidão da Solução:

Garante uma solução ótima global, explorando todas as possíveis combinações de itens.

Eficiência:

Ineficiente para instâncias grandes do problema. Recalcula subproblemas, resultando em repetição de trabalho.

Heurística Gulosa

Complexidade de Tempo:

Linearithmic: $O(n \log n)$, onde n é o número de itens. A ordenação dos itens baseada em uma métrica (razão valor/peso) é a operação mais custosa.

Exatidão da Solução:

Não garante uma solução ótima global. Faz escolhas locais ótimas em cada etapa.

Eficiência:

Eficiente para instâncias grandes do problema. Não recalcula subproblemas, evitando repetição de trabalho.

Resumo das Diferenças:

- A **Complexidade de Tempo** da abordagem recursiva é exponencial, enquanto a heurística gulosa é mais eficiente, sendo linearithmic.

- A **Exatidão da Solução** da abordagem recursiva garante uma solução ótima global, explorando todas as combinações possíveis. A heurística gulosa faz escolhas locais ótimas, buscando soluções aproximadas e rápidas.

- A **Eficiência** da heurística gulosa é mais notável para instâncias grandes do problema, evitando recalculos desnecessários.

- **Aplicabilidade:** A abordagem recursiva é mais adequada para fins educacionais e pequenas instâncias do problema. A heurística gulosa é preferida em situações práticas onde a solução exata é impraticável ou desnecessária, sendo suficiente uma solução aproximada.

REFERÊNCIAS

ANIBALAZEVEDO. *Aula 31 Formulação Matemática do Problema do Caixeiro Viajante*. 2021. Disponível em: <https://www.youtube.com/watch?app=desktop&v=GAiiE8rMzM4&ab_channel=AnibalAzevedo>. Acesso em: 12 dez. 2023.

CLOJURE. *Justificativa*. 2008–2022. Disponível em: <https://clojure.org/about/rationale>. Acesso em: 11 dez. 2023.

CORMEN, Thomas H. et al. *Introduction to Algorithms*. 3rd. ed. [S.l.]: MIT Press, 2009. ISBN-13: 978-0262033848.

CORMEN, Thomas H. et al. *Algoritmos - Teoria e Prática*. Edição 3º. Rua Quintana, 753 – 8º andar 04569-011 – Brooklin – São Paulo – SP: Elsevier Editora Ltda, 2012.

CUNHA, Claudio Barbieri da; BONASSER, Ulisses de Oliveira; ABRAHÃO, Fernando Teixeira Mendes. Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. In: *XVI Congresso da Anpet*. [S.l.: s.n.], 2002.

STACKOVERFLOW. *If P = NP, why does P = NP = NP-Complete? [closed]*. 2014. Disponível em: <https://stackoverflow.com/questions/27416362/if-p-np-why-does-p-np-np-complete>. Acesso em: 12 dez. 2023.