Implementação do Método Branch And Bound Para o Problema da Mochila e Problema do Caixeiro Viajante

Manassés Silva dos Santos[1], Rian Galatas Macêdo Brandão[1], Rodrigo do Nascimento Borges[1]

[1]Departamento de Computação – Universidade Federal do Piauí (UFPI)
Teresina – PI – Brasil

{manasses3mm, rian.team22}@gmail.com, r_borges@ufpi.edu.br

1. Introdução

Em Programação Inteira, o método Branch and Bound é uma poderosa técnica usada para resolver problemas de otimização. A ideia básica por trás desse método é dividir o espaço de busca em subespaços menores, ou ramificações, e eliminar aquelas ramificações que não podem conter uma solução melhor do que a melhor encontrada até o momento. Isso é obtido explorando sistematicamente o espaço de busca, mantendo o controle da melhor solução encontrada até o momento e podando ramificações conhecidas por conter soluções piores do que a melhor encontrada.

Dois problemas bem conhecidos que podem ser resolvidos usando o método Branch and Bound são o Problema da Mochila 0-1 e o Problema do Caixeiro Viajante Simétrico.

O Problema da Mochila 0-1 é um problema de otimização no qual um conjunto de itens com pesos e valores variados deve ser colocado em uma mochila com capacidade de peso limitada. O objetivo é maximizar o valor total dos itens que podem ser carregados na mochila sem ultrapassar sua capacidade de peso. A formulação do Problema da Mochila 0-1 consiste em

$$max\, Z = \sum_{i=1}^{n} v_i x_i \; suj\, a \sum_{i=1}^{n} w_i x_i \leq W,\; x_i \in \{0,1\} \forall\, i = 1, \ldots, n \qquad (1)$$

Para uma variável de decisão inteira e binária x_i, que possuirá valor 1 se o *i-ésimo* item for incluído na mochila e 0 caso contrário, em uma mochila com capacidade *W* e *n* itens com peso *w* e valor *v*.

O Problema do Caixeiro Viajante Simétrico é outro problema clássico de otimização em que um vendedor deve visitar um conjunto de cidades, cada uma com uma determinada distância umas das outras, exatamente uma vez e retornar ao seu ponto de partida, minimizando a distância total percorrida. A formulação do Problema do Caixeiro Viajante Simétrico pode ser dada por

$$min\, Z = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij}$$

$$suj.\, a: \sum_{i=1}^{n} x_{ij} = 1 \forall j = 1, \ldots, n \qquad (2)$$

$$\sum_{j=1}^{n} x_{ij} = 1, \forall i = 1, ..., n$$

$$x_{ij} \in 0, 1, \forall i, j = 1, ..., n$$

$$x_{ii} = 0, \forall i = 1, ..., n$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 2, \forall S \subseteq 1, ..., n, S \neq \emptyset$$

Onde x_{ij} é uma variável de decisão binária e inteira que possui valor 1 e o caminho da cidade *i* até a cidade *j* for incluído na rota final, e d_{ij} é a distância existente entre a cidade *i* e a cidade *j*.

2. Descrição da atividade

O trabalho consiste em implementar o método de busca *Branch and Bound* para os problemas da Mochila 0-1 e do Caixeiro Viajante Simétrico.

Os programas devem gerar instâncias aleatórias dos dois problemas e resolvê-las, exibindo ao final o conjunto solução, o valor da função objetivo atingido, o número de subproblemas criados e o tempo computacional.

Deve ser feita uma sequência de testes com diferentes quantidades de itens para o problema da mochila e diferentes quantidades de cidades para o problema do caixeiro viajantes. Os resultados devem ser apresentados em, para cada problema, um gráfico contendo o tempo de execução em função de n, e um gráfico contendo o número de subproblemas em função de n (n = número de cidades/número de itens).

3. Metodologia

Com relação ao Problema da Mochila 0-1, é utilizada a combinação da heurística gulosa com o método do Branch and Bound. A heurística gulosa é aplicada adicionando os itens com maior relação custo-benefício até que a mochila esteja cheia com a finalidade de gerar uma solução inicial, a qual é avaliada em relação aos limites inferior e superior da solução ótima. Se a solução inicial estiver acima do limite inferior, ela é utilizada como limite superior e o algoritmo começa a explorar o espaço de soluções a partir dela. Caso contrário, o subespaço é descartado e a busca continua no próximo subespaço.

O processo de busca é realizado recursivamente (Branch and Bound) até que todas as soluções possíveis tenham sido exploradas ou até que uma solução ótima seja encontrada.

Tratando do Problema do Caixeiro Viajante Simétrico, é utilizada a combinação da técnica Branch and Bound com a 1-árvore, que pode ser descrita da seguinte forma: Inicialmente, uma solução parcial é construída escolhendo uma cidade como a cidade de origem. Em seguida, a técnica Branch and Bound é usada para explorar todas as soluções possíveis a partir dessa solução parcial. Em cada nó da árvore, a técnica 1-árvore é usada para calcular limites superiores e inferiores para o custo de soluções ainda não exploradas, o que ajuda a cortar soluções inviáveis. O processo continua até que todas as soluções possíveis sejam exploradas e a melhor solução seja encontrada.

4. Implementação

Para colocar em prática as estratégias escolhidas na seção anterior e dar forma aos programas desejados, foram feitas duas implementações, cada uma referente a um dos problemas a serem resolvidos. Em ambos os casos foi feito uso da linguagem Python, bem como das bibliotecas *heapq*, *random* e *time*, presentes nesta linguagem. Estas bibliotecas possuem recursos que foram utilizados, respectivamente, para a construção da árvore de busca, geração de instâncias e contagem do tempo de execução.

As subseções a seguir apresentam um pouco sobre o comportamento dos códigos desenvolvidos, bem como algumas características de sua implementação, tais como as funções definidas e a saída do código.

4.2 “mochila.py”

Referindo-se ao Problema da Mochila, este programa define uma classe `Node` que representa um nó na árvore de busca do problema. Cada nó armazena informações sobre o nível do nó na árvore, o valor total e o peso dos itens selecionados até aquele nível, bem como os respectivos índices dos itens selecionados na solução atual.

No programa também é definida uma função de limitante que calcula um limite superior no valor máximo que pode ser obtido de um nó na árvore de busca. O limite é calculado somando o valor dos itens que podem ser adicionados à mochila no nó atual, de acordo com sua relação valor/peso. Se a capacidade da mochila for excedida, o limite é zero.

A função principal `mochila_bnb` implementa o algoritmo branch and bound. Ele começa inicializando um heap com um nó raiz representando a mochila vazia. A execução prossegue selecionando iterativamente um nó com o valor de limite mais alto do heap, expandindo-o adicionando o próximo item e calculando os valores de limite dos nós filhos resultantes. Os nós filhos com valores de limite superiores ao valor máximo atual são adicionados ao heap. O algoritmo termina quando não há mais nós no heap a serem explorados.

O código gera um conjunto de itens aleatórios e executa o solucionador de problemas da mochila neles. São impressos os itens selecionados, o valor máximo, o número de subproblemas explorados e o tempo decorrido.

4.3 “caixeiro.py”

A implementação para o problema do caixeiro viajante também consiste em várias funções e uma classe `Node`, que é usada para representar uma solução parcial do problema, e que contém em seus atributos os valores referentes ao nível atual, a rota atual, o custo atual e o conjunto de cidades não visitadas.

O algoritmo começa gerando uma matriz de distância aleatória que representa as distâncias entre cada par de cidades. A função principal `caixeiro_bnb` cria então o nó raiz da árvore de busca, que corresponde ao estado inicial do problema. O nó raiz é então adicionado a uma estrutura de heap, que é implementada usando o módulo heapq.

Subsequentemente o programa inicia a execução de uma estrutura de repetição na qual seleciona repetidamente o nó com o menor custo da fila de prioridade e o expande gerando todos os nós filhos possíveis. O custo de cada nó filho é calculado

adicionando a distância entre a última cidade visitada e a próxima cidade não visitada ao custo do nó pai.

É possível observar também a implementação de uma função de limite inferior, chamada de `bound`, que tem como objetivo estimar o custo de uma solução parcial. A função constrói uma árvore geradora mínima para o subgrafo de cidades não visitadas e então calcula o custo do caminho mais barato de cada cidade na árvore para o conjunto de cidades não visitadas. A soma do custo da árvore geradora mínima e o custo dos caminhos mais baratos é usado como limitante.

Se o custo de um nó filho for menor que a melhor solução atual, o nó filho é adicionado à fila de prioridade. O algoritmo continua a expandir os nós até que todos os nós da fila tenham sido explorados, momento em que retorna a melhor solução encontrada. O programa então exibe a melhor rota, o custo total, o número de subproblemas explorados e o tempo gasto para resolver o problema. Conseguimos obter um valor total de Z = 19 na mochila.

5. Resultados e Discussão

Para o problema do Caixeiro Viajante, aplicamos um exemplo com matrizes de distâncias [10, 10], conseguimos obter um caminho com o menor custo total, visitando as cidades todas as cidades.

Para a matriz de distâncias:

[[0, 8, 6, 3, 3, 5, 8, 4, 1, 1],

[8, 0, 6, 10, 7, 10, 2, 9, 9, 4],

[6, 6, 0, 2, 2, 3, 7, 2, 3, 4],

[3, 10, 2, 0, 1, 10, 2, 8, 4, 3],

[3, 7, 2, 1, 0, 2, 7, 7, 8, 7],

[5, 10, 3, 10, 2, 0, 8, 8, 9, 4],

[8, 2, 7, 2, 7, 8, 0, 4, 2, 8],

[4, 9, 2, 8, 7, 8, 4, 0, 1, 8],

[1, 9, 3, 4, 8, 9, 2, 1, 0, 4],

[1, 4, 4, 3, 7, 4, 8, 8, 4, 0]]

Obtivemos os seguintes resultados:

Rota: [0, 8, 7, 2, 5, 4, 3, 6, 1, 9, 0]

Z: 19

Subproblemas: 11301

Tempo decorrido: 0.065001726151 segundos.

Já para a matriz de distâncias:

[[0, 3, 6, 8, 5, 8, 1, 10, 10, 2],

[3, 0, 4, 10, 4, 7, 1, 3, 5, 1],

[6, 4, 0, 2, 6, 10, 5, 9, 8, 5],

[8, 10, 2, 0, 5, 1, 8, 8, 2, 4],

[5, 4, 6, 5, 0, 2, 10, 7, 3, 7],

[8, 7, 10, 1, 2, 0, 10, 7, 8, 8],

[1, 1, 5, 8, 10, 10, 0, 1, 3, 7],

[10, 3, 9, 8, 7, 7, 1, 0, 9, 1],

[10, 5, 8, 2, 3, 8, 3, 9, 0, 9],

[2, 1, 5, 4, 7, 8, 7, 1, 9, 0]]

Obtivemos os resultados seguintes:

Rota: [0, 9, 7, 6, 8, 4, 5, 3, 2, 1, 0]
Z: 22
Subproblemas: 5729
Tempo decorrido: 0.050996541977 segundos

Para o Problema da Mochila, ao aplicarmos o código em um exemplo com 1000 objetos com uma capacidade de 100, conseguimos obter um valor total de Z na mochila, sendo que o objeto A, com peso P e valor V, foi inserido na mochila, assim como outros objetos.

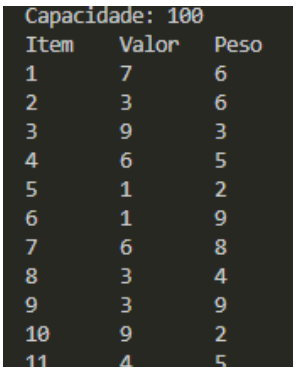


Figura 1: Exemplo de instância do Problema da Mochila 0-1.

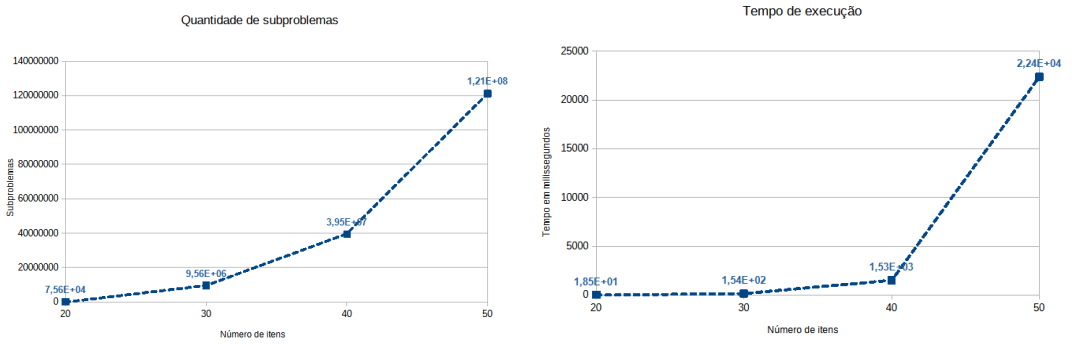
Obtivemos os seguintes resultados:

Itens: [1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 18, 19, 20, 21, 25, 26, 27, 28, 30, 31, 34, 35, 39, 41, 56]

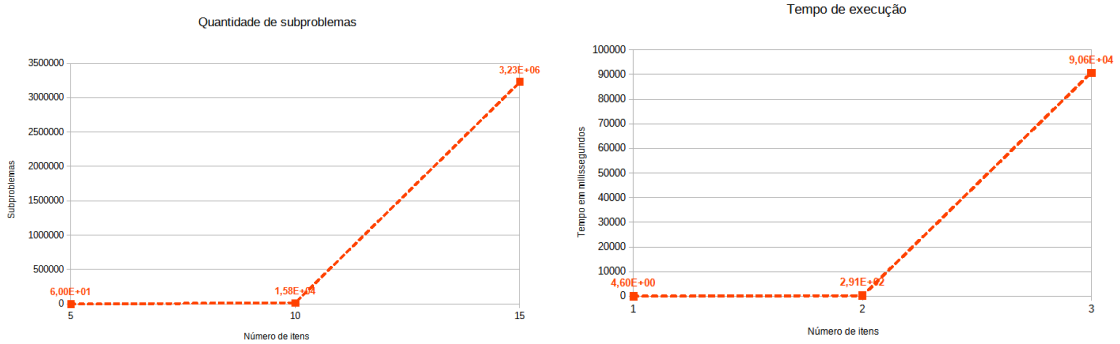
Z: 170

Subproblemas: 61225

Tempo decorrido: 0.140090942383 segundos.



Figuras 2 e 3: Gráficos de quantidade de subproblemas e tempo de execução para o Problema da Mochila.



Figuras 4 e 5: Gráficos de quantidade de subproblemas e tempo de execução para o Problema do Caixeiro Viajante.

Esses resultados são comparáveis com soluções conhecidas para o problema do caixeiro viajante e problema da mochila, validando a eficácia do código desenvolvido.

6. Conclusão

Os resultados obtidos demonstraram a eficácia dos códigos, comparáveis com soluções conhecidas para os problemas da mochila e do caixeiro viajante. No entanto, também foram apresentadas limitações dos códigos, tanto em termos de restrições quanto em termos de limitações computacionais para problemas maiores.

Dessa forma, concluímos que a programação inteira é uma área de grande importância para a resolução de problemas de otimização em diversas áreas. Os códigos desenvolvidos neste relatório são exemplos de como a programação inteira pode ser aplicada para encontrar soluções ótimas para problemas complexos, contribuindo para a melhoria de processos e tomadas de decisão.

Referências

Bunescu, R. (n.d.). Design and Analysis of Algorithms: Lecture 16. Disponível em: https://webpages.charlotte.edu/rbunescu/courses/ou/cs4040/lecture16.pdf. Acesso em Março de 2023.

Munoz de Cote, E., & Mariano, A. (2021). TSP Formulations. Disponível em: https://dm872.github.io/assets/dm872-TSP_Formulations.pdf. Acesso em Março de 2023.

Kianfar, Kiavash. (2011). Branch‐and‐Bound Algorithms. 10.1002/9780470400531.eorms0116.

GeeksforGeeks. (n.d.). Implementation of 0/1 Knapsack using Branch and Bound. Disponível em: https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/. Acesso em Março de 2023.

GeeksforGeeks. (n.d.). Traveling Salesman Problem using Branch and Bound. Disponível em: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/. Acesso em Março de 2023.

Garey, M. R.; Johnson, D.S. Computers and intractability: a guide to the theory of NP-completeness. Freeman. 1979.