Algoritmos 2
Trabalho Prático 2
Backtracking e Branch and Bound

Otávio de Meira Lima - 2019054900

DCC - Universidade Federal de Minas Gerais
Professor Renato Vimieiro

Abstract: This document refers to practical work 2 of the Algorithms 2 course at the Federal University of Minas Gerais, with details of the implementation of each algorithm used and analysis of execution time.

Resumo: Este documento se refere ao trabalho prático 2 da disciplina Algoritmos 2 da Universidade Federal de Minas Gerais, possuindo detalhamento da implementação de cada algoritmo utilizado e análise de tempo de execução.

1. Introdução

Esse trabalho prático consiste em implementar o problema da mochila utilizando Backtracking e Branch and Bound, métodos para soluções de problemas difíceis que usualmente possuem uma abordagem de força bruta. Essas soluções criam árvores binárias, onde cada nó é uma solução diferente. Seu custo é bem maior do que outras abordagens, como programação dinâmica, já que todas as soluções válidas são avaliadas a fim de encontrar a solução ótima.

2. Detalhes da implementação

Os algoritmos, leitura dos dados e cálculo dos resultados foram implementados em Python.

Para adquirir os resultados, basta executar o arquivo main.py, onde é calculado o tempo de execução e a resposta para cada arquivo teste por meio do comando:

python .\main.py

Os resultados ficam armazenados no arquivo results.csv.

3. Backtracking

Backtracking é um tipo de algoritmo onde diversas soluções do problema são examinadas até que se encontre uma solução ótima. Dessa forma, a solução é encontrada por meio de força bruta. No caso do problema da mochila, várias combinações de itens são inspecionadas até que não exista mais uma combinação válida, ou seja, aquelas que a soma do peso dos itens ultrapassa o peso total da mochila.

Mesmo que a solução ótima já esteja calculada, o algoritmo continua à procura de outras combinações que possam resultar em um valor maior. Consequentemente, o algoritmo fica com o custo muito mais alto do que implementações do problema otimizadas, como programação dinâmica.

Em um conjunto com n itens, o algoritmo no pior caso executa em tempo da ordem de $O(n^2)$, já que todas as soluções podem ser avaliadas.

```python
def backtracking_knapsack(n, w, weights, values):

    if (n == 0 or w == 0):
        return 0

    if (weights[n-1] > w):
        return backtracking_knapsack(n-1, w, weights, values)

    return max(values[n-1] + backtracking_knapsack(n-1, w-weights[n-1], weights, values),
                   backtracking_knapsack(n-1, w, weights, values))
```

Sua implementação é por meio da recursividade. No caso base, caso o peso de um item seja maior do que a mochila pode carregar, é retornado 0. No passo recursivo, é retornado o valor máximo entre adicionar o item à mochila e não adicioná-lo.

4. Branch and Bound

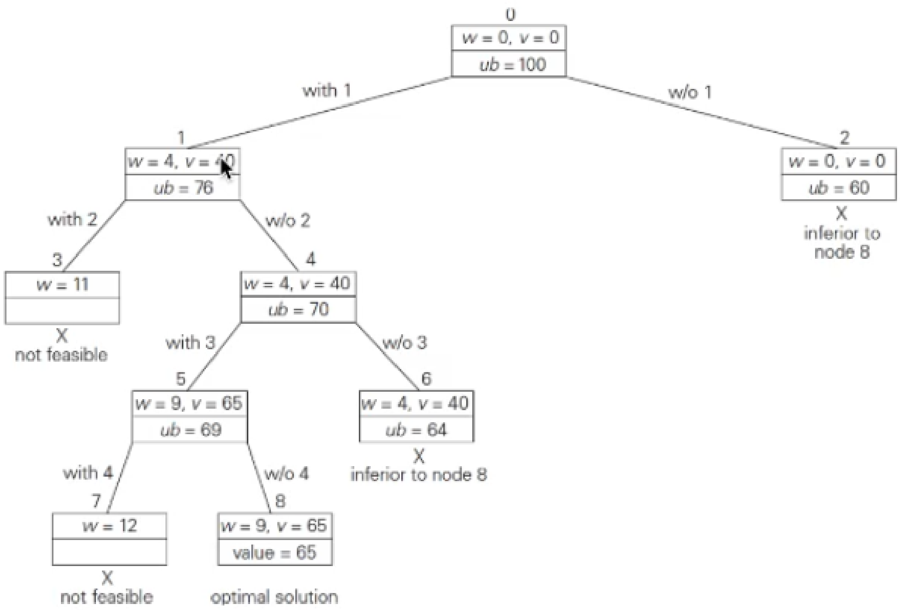
A implementação do algoritmo Branch and Bound para solucionar o problema da mochila é semelhante ao backtracking, porém de forma otimizada. A cada nó da árvore binária, criamos uma estimativa de valor para qual aquela ramificação pode alcançar.

Para calcular essa estimativa, os itens devem ser ordenados pela razão valor/peso. Dessa forma, os nós dos níveis mais baixos vão calcular estimativas com os itens de maior razão do valor por peso.

Podemos exemplificar por meio da seguinte instância do problema.

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

W=10



É possível notar que o primeiro nó da árvore é uma combinação onde nenhum dos itens foi inserido na mochila.

A cada nível da árvore, é calculado o valor total inserindo (nó da esquerda) ou não inserindo o item atual. Caso a estimativa feita for menor que a máxima calculada, pode-se concluir que aquela ramificação não contém uma solução melhor que a máxima já calculada, já que, como os itens estão ordenados, não é possível completar a mochila com itens de menor razão valor/peso e ultrapassar o valor máximo já calculado.

Com essa otimização, é possível deixar de calcular combinações que garantidamente possuem solução pior do que a ótima.

```python
def branch_and_bound_knapsack(W, values, weights, n):

    values, weights = sort_values_and_weights(values, weights, n)

    pq = []

    v = Node(-1, 0, 0)
    maxProfit = 0
    v.bound = bound(v, n, W, values, weights)

    pq.append(v)

    while pq:

        pq.sort(key=lambda i: i.bound)

        v = pq.pop(0)

        if v.bound > maxProfit:

            u = Node(0, 0, 0)

            u.level = v.level + 1
            u.value = v.value + values[u.level]
            u.weight = v.weight + weights[u.level]

            u.items = v.items.copy()
            u.items.append(u.level)

            if u.weight <= W and u.value > maxProfit:
                maxProfit = u.value

            u.bound = bound(u, n, W, values, weights)

            if u.bound > maxProfit:
                pq.append(u)

            u2 = Node(u.level, v.value, v.weight)
            u2.bound = bound(u2, n, W, values, weights)
            u2.items = v.items.copy()

            if u2.bound > maxProfit:
                pq.append(u2)

    return maxProfit
```

O algoritmo mantém uma fila de prioridades que possui o nó de maior estimativa na primeira posição. A cada iteração, um nó é removido da fila e calcula-se o valor total adicionando certo item. Isso é feito até que a fila esteja vazia. Durante toda a iteração, mantém-se uma variável denominada **maxProfit** que contém a soma dos valores da melhor combinação de itens adicionados na mochila até então.

5. Análise do Tempo de Execução

A fim de comparação do tempo de execução de cada algoritmo, cada execução dos arquivos testes foram armazenados no arquivo results.csv de forma automática por meio da biblioteca **csv**. Nele, é possível saber o nome do arquivo, tempo de execução por Backtracking, tempo de execução por Branch and Bound e o valor máximo que a mochila pode carregar.

Como o algoritmo branch and bound possui uma otimização por meio de cálculo de estimativas, é clara a diferença no tempo de execução em cada arquivo teste. Porém, apenas o arquivo f8_l-d_kp_23_10000 possui um tempo de execução menor na execução do backtracking. Em análises mais profundas, podemos perceber que a fila de prioridades calculada nesse caso aumenta o tempo de execução.

Como o número de itens possíveis a serem adicionados na mochila é pequeno, é possível perceber que ambas as implementações executam em menos de 1 segundo em sua maioria. A percepção da otimização do algoritmo branch and bound se dá na casa dos milésimos.

```
Arquivo;Tempo de Execucao por Backtracking;Tempo de Execucao por Branch and Bound;Resultado
f6_l-d_kp_10_60;0.00048804283142089844;0.0011799335479736328;52.0
f1_l-d_kp_10_269;0.00048232078552246094;0.0006673336029052734;295.0
f9_l-d_kp_5_80;2.574920654296875e-05;0.00010466575622558594;130.0
f10_l-d_kp_20_879;0.8031282424926758;0.005913972854614258;1025.0
f5_l-d_kp_15_375;0.014988183975219727;0.0018393993377685547;481.069368
f7_l-d_kp_7_50;6.961822509765625e-05;0.00011897087097167969;107.0
f8_l-d_kp_23_10000;3.91336989402771;17.09037208557129;9767.0
f2_l-d_kp_20_878;0.801020622253418;0.005873680114746094;1024.0
f4_l-d_kp_4_11;1.33514404296875e-05;0.00011682510375976562;23.0
f3_l-d_kp_4_20;1.3113021850585938e-05;8.082389831542969e-05;35.0
```

Ao executar os algoritmos no terminal do Windows, o tempo de execução em vários arquivos teste estava retornando 0.0 segundos. Para contornar esse problema, o mesmo código foi executado por meio de um notebook no Google Colab, disponibilizado para leitura no link a seguir:

https://colab.research.google.com/drive/1XKgPKB3DqyvF3ha6FaRE-SIFc9OoEQGg?usp=sharing

6. Referências

Algoritmos: teoria e prática.
T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein.
Editora Campus.

Introduction to Algorithms. T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein. 3rd Edition. MIT Press.