Análise de Desempenho de Algoritmos para o Problema da Mochila 0-1

Felipe Sasdelli - 18.2.4002
Gabriel Negri -19.1.4976

19 de novembro de 2023

1 Resumo

Este trabalho tem como objetivo realizar uma análise de desempenho de diferentes algoritmos para resolver o Problema da Mochila 0-1. O problema consiste em determinar a melhor combinação de itens para colocar em uma mochila, maximizando o valor total, dado um limite de peso. Os algoritmos analisados incluem Programação Dinâmica, Backtracking e Branch-and-Bound, todos implementados em linguagem C, visando um desempenho computacional superior.

Também foi realizada uma avaliação experimental a fim de comparar o desempenho desses algoritmos em diferentes cenários, variando o peso e a quantidade de itens da mochila. Através da análise de diversas instâncias e a aplicação de testes estatísticos t pareado com 95% de confiança, buscamos determinar qual dos algoritmos apresenta o melhor desempenho em termos de tempo de execução e precisão. A apresentação dos resultados é auxiliada pelo uso de Python para a geração de gráficos e tabelas, facilitando a interpretação e comparação dos dados obtidos.

2 Introdução

A investigação do Problema da Mochila 0-1 é relevante tanto para o campo acadêmico quanto para aplicações industriais, onde a escolha do algoritmo de resolução pode impactar diretamente na eficiência e viabilidade de operações. A Programação Dinâmica, o Branch and Bound e o Backtracking são técnicas que oferecem diferentes compromissos entre tempo de execução e consumo de memória, tornando-se essencial avaliar suas performances sob condições variadas. Neste trabalho, essas abordagens são implementadas em C, a fim de explorar o potencial da linguagem em termos de otimização de recursos e tempo de processamento. A escolha de C está alinhada com a necessidade de lidar com a grande escala dos dados de entrada, que impõe desafios significativos em termos de eficiência algorítmica.

Para aprofundar a análise, foram criadas instâncias de teste que variam desde $n = 25$ até $n = 52,428,800$ itens, com capacidades de mochila correspondentes de $W = 100$ até $W = 52,428,800$, mantendo n fixo em 400 para uma série de testes dedicados. Cada configuração foi avaliada repetidas vezes para assegurar a confiabilidade estatística dos resultados relacionados ao tempo de execução.

2.1 Repositório no GitHub

Todos os arquivos gerados para a realização deste trabalho estão presentes no arquivo enviado na plataforma Moodle, e também no seguinte repositório do GitHub:

`https://github.com/felipeperet/knapsack-c`

2.2 Estrutura do Projeto

Segue a tabela de como foi configurado o respectivo projeto:

```
knapsack/
 bin/       - Executáveis compilados
 src/       - Códigos fonte dos algoritmos
 include/   - Arquivos cabeçalho
 obj/       - Arquivos objeto gerados na compilação
 data/      - Entradas e saídas dos testes
 Makefile   - Script para automatizar a compilação
 LICENSE.md - Licença do projeto
 README.md  - Informações e instruções sobre o projeto
```

3 Algoritmos Analisados

3.1 Programação Dinâmica (DP)

Descrição: A abordagem de Programação Dinâmica para o problema da mochila 0-1 visa calcular o valor máximo que pode ser carregado na mochila sem ultrapassar o peso permitido. Ele constrói uma matriz K onde cada célula K[i][w] representa o valor máximo que pode ser alcançado com i itens e uma capacidade máxima de w. O algoritmo preenche essa matriz de maneira bottom-up, garantindo que todas as sub-soluções necessárias sejam calculadas antes da solução final.

- **Código:**

```c
// Implementação do problema da mochila 0-1 utilizando programação dinâmica.
int knapsack_dynamic(int W, int wt[], int val[], int n) {
    int i, w;
    int **K = allocate_matrix(n + 1, W + 1);

    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] =
                    max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                   K[i][w] = K[i - 1][w];
        }
    }

    int result = K[n][W];

    deallocate_matrix(K, n + 1);

    return result;
}
```

- **Funcionamento do algoritmo:**

A matriz K é inicializada com zeros para a linha 0 e a coluna 0, representando o caso base onde não há itens ou a capacidade da mochila é 0. O uso da função 'max' determina o maior valor

entre incluir o item atual ou não, otimizando o valor total na mochila até o peso atual w. Após o cálculo, a matriz K é desalocada para liberar a memória alocada. A implementação assume que uma função auxiliar 'allocate-matrix' é usada para alocar dinamicamente a matriz, e 'deallocate-matrix' para liberá-la.

- **Análise de Complexidade:**

A complexidade de tempo do algoritmo é $O(nW)$, onde n é o número de itens e W é a capacidade máxima da mochila. Isso ocorre porque o algoritmo itera através de todas as combinações possíveis de n itens e capacidades de mochila até W. A complexidade de espaço também é $O(nW)$, devido à matriz K que armazena os resultados intermediários.

3.2 Backtracking

Descrição: O algoritmo de Backtracking aborda o problema da mochila 0-1 por meio da exploração recursiva de todas as combinações possíveis de itens. Ele avalia a inclusão e a exclusão de cada item, navegando por uma árvore de decisões até que todas as possibilidades sejam consideradas ou uma condição de parada seja atingida, como ultrapassar o peso permitido da mochila ou considerar todos os itens.

- **Código:**

```
// Implementação do problema da mochila 0-1 utilizando backtracking.
int knapsack_backtracking(int W, int wt[], int val[], int n, int idx,
                          int total_val, int total_wt) {
    if (total_wt > W || idx == n)
        return total_val;

    int value_with = 0;
    if (total_wt + wt[idx] <= W) {
        value_with = knapsack_backtracking(
            W, wt, val, n, idx + 1, total_val + val[idx], total_wt + wt[idx]);
    }

    int value_without =
        knapsack_backtracking(W, wt, val, n, idx + 1, total_val, total_wt);

    return max(value_with, value_without);
}
```

- **Funcionamento do algoritmo:**

O backtracking funciona como uma busca em profundidade na árvore de decisões do problema. Em cada etapa, o algoritmo considera incluir o item atual na mochila ('value-with') ou excluir ('value-without'), sempre checando se a capacidade total não é excedida. O processo se repete até que todos os itens sejam considerados, retornando o valor máximo obtido sem exceder o peso da mochila.

- **Análise de Complexidade:**

A complexidade de tempo do algoritmo de Backtracking é $O(2^n)$ no pior caso, onde n é o número de itens, pois em teoria, cada item tem duas possibilidades: ser incluído ou excluído da mochila. Isso leva a uma árvore de decisão binária, onde todas as possíveis combinações de itens são exploradas. A complexidade de espaço é O(n) devido à profundidade da pilha de chamadas recursivas, que corresponde ao número de itens a serem considerados.

3.3 Branch-and-Bound

Descrição: O Branch-and-Bound para o problema da mochila 0-1 é um método de otimização que sistematicamente divide o espaço de busca para resolver problemas de decisão. Este algoritmo usa uma fila dinâmica para gerenciar os nós de busca e uma função de cota superior 'bound' para avaliar o potencial máximo de lucro de cada nó. A utilização da fila dinâmica é essencial para gerenciar o uso de memória de forma eficiente, pois permite que o algoritmo processe um grande número de possibilidades sem alocar espaço para todas elas antecipadamente.

- **Código:**

```
// Implementação do problema da mochila 0-1 utilizando Branch-and-Bound.
int knapsack_branch_and_bound(int W, Item arr[], int n) {
    qsort(arr, n, sizeof(Item), cmp);

    DynamicQueue queue;
    DynamicQueue_init(&queue, 1024);

    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    u.bound = bound(u, n, W, arr);
    DynamicQueue_push(&queue, u);

    int maxProfit = 0;

    while (queue.size) {
        u = DynamicQueue_pop(&queue);

        if (u.level == n - 1 || u.bound <= maxProfit) {
            continue;
        }

        v.level = u.level + 1;

        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;
        if (v.weight <= W && v.profit > maxProfit) {
            maxProfit = v.profit;
        }
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit) {
            DynamicQueue_push(&queue, v);
        }

        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit) {
            DynamicQueue_push(&queue, v);
        }
    }

    DynamicQueue_free(&queue);
    return maxProfit;
}
```

- **Funcionamento do algoritmo:**

O algoritmo Branch-and-Bound inicia ordenando os itens pela razão valor/peso. A fila dinâmica é usada para armazenar os nós que representam soluções parciais. Cada nó contém o nível do item que está sendo considerado, o lucro acumulado, o peso total e a cota 'bound', que estima o lucro máximo que pode ser alcançado. Se o peso acumulado de um nó excede a capacidade da mochila, ele é descartado; caso contrário, o nó é expandido, criando novos nós com e sem o próximo item, e a cota é calculada para determinar se o nó deve ser explorado mais adiante.

- **Função 'bound':**

```
float bound(Node u, int n, int W, Item arr[]) {
    if (u.weight >= W) {
        return 0;
    }

    float result = (float)u.profit;
    int totweight = u.weight;

    for (int j = u.level + 1; j < n; j++) {
        if (totweight + arr[j].weight <= W) {
            totweight += arr[j].weight;
            result += arr[j].value;
        } else {
            result += (W - totweight) * (float)arr[j].value / arr[j].weight;
            break;
        }
    }

    return result;
}
```

A função 'bound' é usada para calcular a cota superior do lucro que pode ser alcançado a partir de um nó. Ela adiciona ao lucro atual os valores dos itens subsequentes até que o peso adicional exceda a capacidade da mochila. Nesse ponto, ela adiciona uma fração do último item que caberia na mochila para completar a capacidade, fornecendo uma estimativa otimista do lucro máximo possível.

- **Análise de Complexidade:**

O algoritmo Branch-and-Bound possui uma complexidade de tempo que varia significativamente com a eficácia da poda realizada pela função 'bound'. Idealmente, a função 'bound' permite podar o espaço de busca de forma que o algoritmo evite a exploração de nós que não levariam a uma solução ótima, reduzindo assim a complexidade de tempo média para melhor que exponencial. No entanto, na pior das hipóteses, se a poda for ineficaz, o algoritmo pode explorar um espaço de busca exponencial, tornando-se $O(2^n)$. A complexidade de espaço é determinada pelo número de nós na fila dinâmica, que, no pior caso, pode crescer até o tamanho total do espaço de estado, mas geralmente é gerenciada de forma eficiente para manter a complexidade em $O(b^d)$, com b representando o fator de ramificação e d a profundidade da solução mais profunda.

4 Avaliação Experimental

Conduzimos dois conjuntos de experimentos para avaliar a performance dos algoritmos de Programação Dinâmica, Backtracking e Branch-and-Bound. No primeiro conjunto, variamos o número de itens n em potências de dois, começando com $n = 25$. No segundo conjunto, mantivemos o número de itens constante em $n = 400$ e variamos a capacidade da mochila W, também em potências de dois a partir de $W = 100$. Analisamos o tempo de execução dos algoritmos em ambas as configurações e observamos o impacto no desempenho conforme aumentamos n e W, respectivamente.

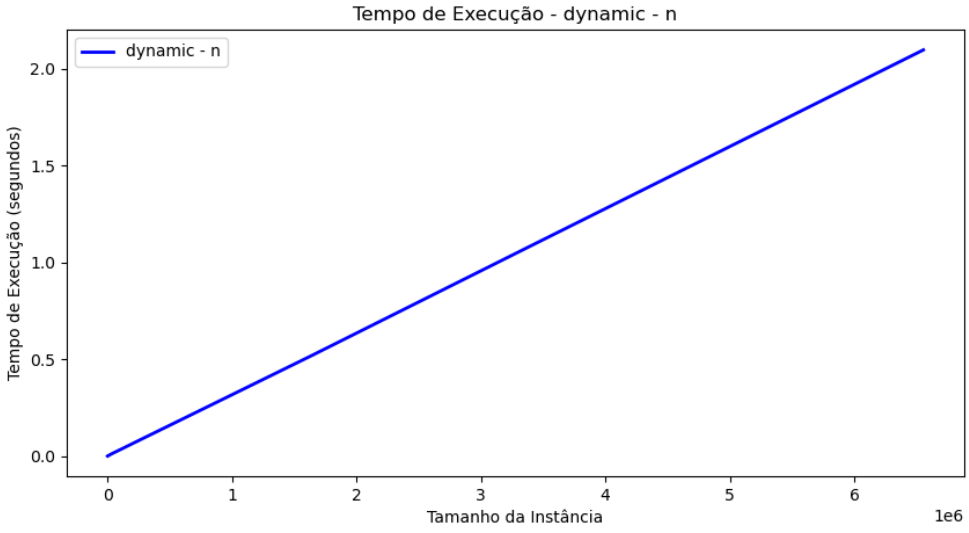
4.1 Configuração da Máquina

- Sistema Operacional: Linux (NixOS)
- Processador: I7 de 11ª geração
- Memória RAM: 16GB

4.2 Gráficos de Tempo de Execução

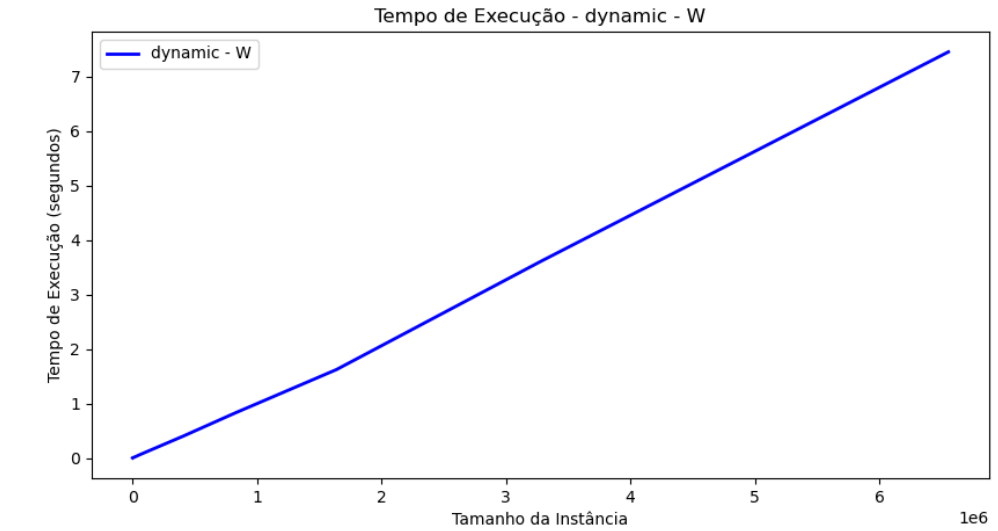
4.2.1 Programação Dinâmica (DP)

- **Programação Dinâmica com variação de n:**



O gráfico demonstra um aumento linear no tempo de execução conforme o número de itens n cresce. Isso está alinhado com a complexidade de tempo $O(nW)$ da Programação Dinâmica, pois apenas o número de itens está aumentando, enquanto a capacidade da mochila é mantida constante.

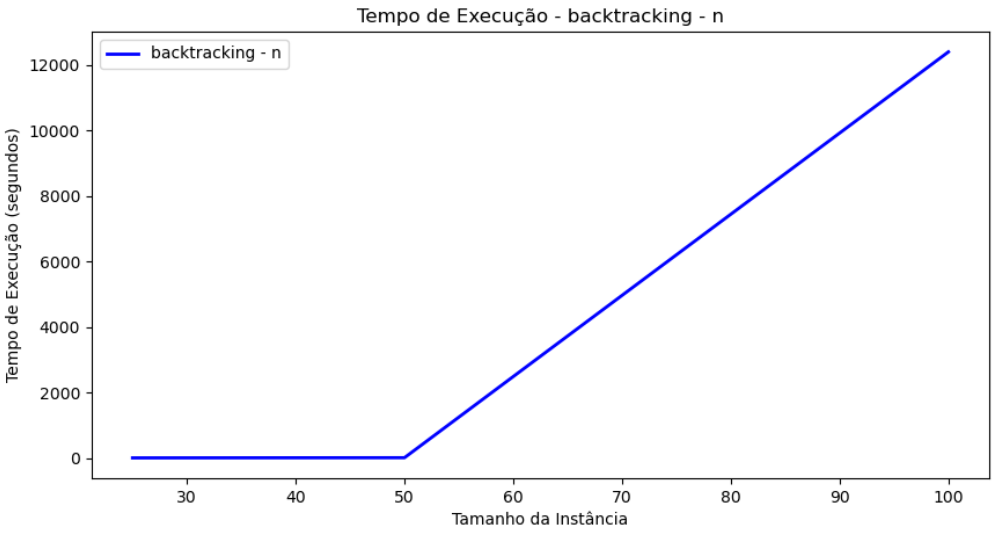
- **Programação Dinâmica com variação de W:**



Similarmente, vemos um aumento linear no tempo de execução à medida que a capacidade da mochila W aumenta. A tendência linear está de acordo com a complexidade esperada $O(nW)$ da Programação Dinâmica, indicando um comportamento previsível e eficiente desse algoritmo.

4.2.2 Backtracking

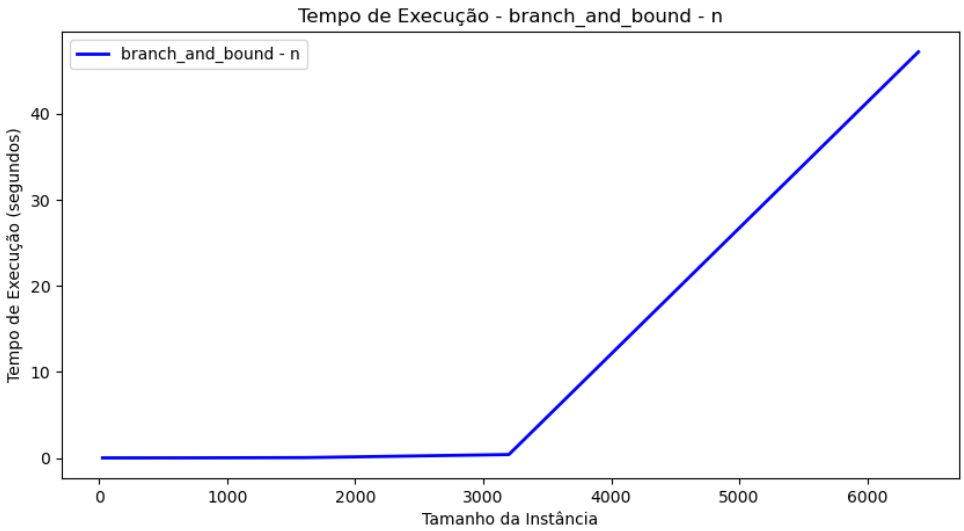
- **Backtracking com variação de n:**



O gráfico mostra um aumento exponencial no tempo de execução à medida que o número de itens n aumenta. Isso é esperado, uma vez que o algoritmo de Backtracking tem uma complexidade de tempo teórica de $O(2^n)$, refletindo a natureza de sua busca exaustiva por todas as possíveis combinações de itens.

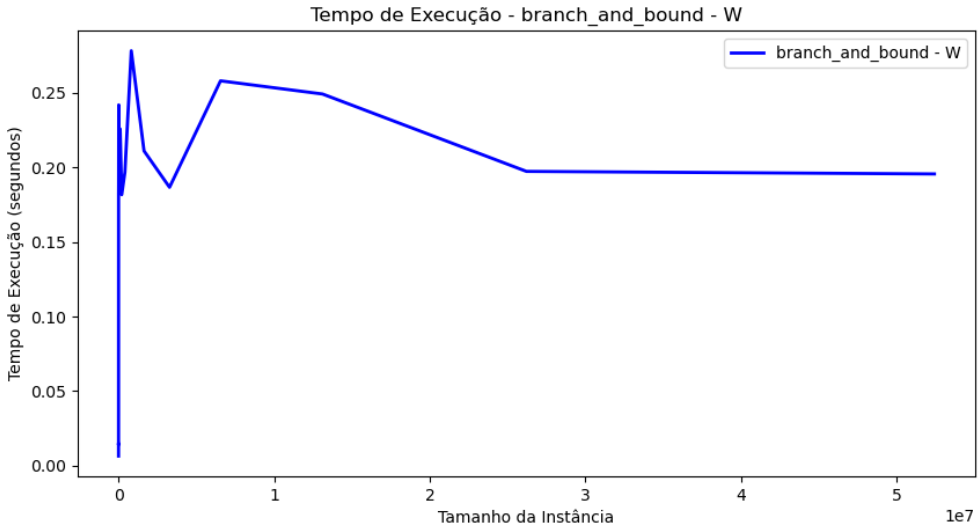
4.2.3 Branch-and-bound

- **Branch-and-Bound com variação de n:**



Observamos um aumento polinomial no tempo de execução, o que é um comportamento esperado para o Branch-and-Bound. Este algoritmo tende a ser mais eficiente que o Backtracking devido à sua capacidade de podar o espaço de busca, o que é evidenciado pelo crescimento mais suave em comparação com o Backtracking.

- **Branch-and-Bound com variação de W:**



O gráfico apresenta um comportamento inicialmente instável, que se estabiliza para valores maiores de W. Isso pode indicar que, para capacidades maiores da mochila, a função de cota (bound) consegue identificar rapidamente soluções promissoras, diminuindo a necessidade de explorar muitos nós.

5 Testes de Paridade (95% de confiança)

5.1 Backtracking - 'n variando'

Tamanho	Tempo (segundos)
25	0.0058
50	6.0237
100	12390.9100
Média	4132.3132
Lim. Inf.	-13634.6255
Lim. Sup.	21899.2519

Os resultados mostram um grande intervalo no tempo de execução para diferentes tamanhos de instância. A média é positivamente distorcida pelo caso de tamanho 100.

5.2 Branch and Bound - 'n variando'

Tamanho	Tempo (segundos)
25	0.0002
50	0.0007
100	0.0019
200	0.0011
400	0.0023
800	0.0059
1600	0.0333
3200	0.3901
6400	47.2155
Média	5.2946
Lim. Inf.	-6.7896
Lim. Sup.	17.3787

A execução da abordagem Branch and Bound mostra um aumento exponencial do tempo com o aumento do tamanho da instância.

5.3 Branch and Bound - 'W variando'

Tamanho	Tempo (segundos)
100	0.0064
200	0.0148
400	0.0141
800	0.0342
1600	0.1003
3200	0.1818
6400	0.1873
12800	0.2419
25600	0.1895
51200	0.1926
102400	0.2258
204800	0.1817
409600	0.1968
819200	0.2783
1638400	0.2111
3276800	0.1867
6553600	0.2581
13107200	0.2492
26214400	0.1973
52428800	0.1956
Média	0.1672
Lim. Inf.	0.1273
Lim. Sup.	0.2071

Os tempos para Branch and Bound 'W' variam menos em relação ao tamanho da instância do que na versão 'n', indicando uma maior previsibilidade.

5.4 Dynamic - 'n variando'

Tamanho	Tempo (segundos)
25	0.0000
50	0.0001
100	0.0001
200	0.0003
400	0.0004
800	0.0002
1600	0.0004
3200	0.0008
6400	0.0017
12800	0.0035
25600	0.0082
51200	0.0166
102400	0.0324
204800	0.0649
409600	0.1296
819200	0.2588
1638400	0.5175
3276800	1.0448
6553600	2.0960
Média	0.2198
Lim. Inf.	-0.0323
Lim. Sup.	0.4719

A abordagem dinâmica exibe um padrão consistente de aumento de tempo com o aumento do tamanho da instância, com um intervalo de confiança estreito.

5.5 Dynamic - 'W variando'

Tamanho	Tempo (segundos)
100	0.0005
200	0.0008
400	0.0016
800	0.0025
1600	0.0017
3200	0.0060
6400	0.0078
12800	0.0125
25600	0.0244
51200	0.0520
102400	0.1023
204800	0.2023
409600	0.4024
819200	0.8205
1638400	1.6225
3276800	3.6061
6553600	7.4559
Média	0.8425
Lim. Inf.	-0.1536
Lim. Sup.	1.8385

Para a abordagem dinâmica com arquivos terminando em 'W', o tempo de execução aumenta com o tamanho da instância, mas a taxa de crescimento parece diminuir para instâncias muito grandes.

6 Conclusão

A análise de desempenho dos algoritmos Programação Dinâmica, Backtracking e Branch-and-Bound para resolver o Problema da Mochila 0-1 revelou diferenças significativas em seus tempos de execução e comportamentos em resposta a variações no número de itens e capacidades da mochila. A Programação Dinâmica mostrou-se eficiente e previsível, com um aumento linear no tempo de execução, o que é consistente com sua complexidade teórica de $O(nW)$. O Backtracking, embora simples em sua implementação, provou ser impraticável para instâncias maiores devido ao seu tempo de execução exponencial. O Branch-and-Bound ofereceu um meio-termo, superando o Backtracking e, em muitos casos, aproximando-se da eficiência da Programação Dinâmica.

Os testes de paridade realizados com 95% de confiança ajudaram a confirmar essas observações, demonstrando a superioridade da Programação Dinâmica em termos de tempo de execução, especialmente para grandes capacidades de mochila. No entanto, para pequenas instâncias, o Branch-and-Bound e até mesmo o Backtracking podem ser considerados devido à sua menor sobrecarga inicial. A escolha do algoritmo, portanto, deve ser informada pelo tamanho da instância e pelos recursos disponíveis.

7 Referências Bibliográficas:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
- Martello, S., & Toth, P. (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons.