Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Ciência da Computação - DECOM

Trabalho Prático
O problema da mochila 0-1

BCC241 - Projeto e Análise de Algoritmos

Kayo Xavier Nascimento Cavalcante Leite - 21.2.4095

Lavínia Fonseca Pereira - 19.1.4170

Thaís Ferreira de Oliveira Almeida - 21.2.4012

Professor:Anderson Almeida Ferreira

Ouro Preto
24 de março de 2025

Conteúdo

1 Introdução

O problema da mochila 0-1 é um problema clássico de otimização combinatória amplamente estudado na área de ciência da computação. Dado um conjunto de n itens, onde cada item i possui um peso w_i e um valor v_i, e uma mochila com capacidade máxima W, o objetivo é selecionar um subconjunto de itens que maximize o valor total, sem exceder a capacidade da mochila. A restrição 0-1 significa que cada item pode ser incluído no máximo uma vez (ou seja, não é permitido incluir frações de itens). Este problema é NP-difícil, o que motiva o desenvolvimento de algoritmos exatos e heurísticos para resolvê-lo.

Este trabalho tem como objetivo implementar e avaliar empiricamente três algoritmos exatos para o problema da mochila 0-1: Programação Dinâmica (DP), Backtracking (BT) e Branch-and-Bound (BB). A avaliação foi conduzida por meio de dois experimentos distintos:

- **Experimento 1 (n variando):** Fixou-se a capacidade $W = 100$ e variou-se o número de itens n de 10 a 25.600, dobrando o valor a cada iteração (10, 20, 30, ..., 25.600).
- **Experimento 2 (W variando):** Fixou-se $n = 800$ (inicialmente planejado como $n = 400$, mas ajustado para permitir instâncias maiores, já que com $n = 400$ só foi possível rodar até $W = 6.400$) e variou-se W de 100 a 25.600, também dobrando o valor a cada iteração (100, 200, 400, ..., 25.600).

Para cada configuração, foram geradas 20 instâncias aleatórias, com pesos w_i entre 1 e 30 e valores v_i entre 1 e 100. A métrica principal de avaliação foi o tempo médio de execução (em segundos), e os resultados foram analisados estatisticamente com o teste t pareado a 95% de confiança. Os tempos médios foram plotados em gráficos com intervalos de confiança para facilitar a visualização do desempenho dos algoritmos.

Os resultados mostram que o Branch-and-Bound apresenta o melhor desempenho geral, especialmente no experimento com W variando, onde seu tempo de execução permanece praticamente constante. O Backtracking, por outro lado, torna-se inviável para $n > 50$, devido à sua complexidade exponencial. A Programação Dinâmica é eficiente para valores pequenos de W, mas seu tempo de execução cresce linearmente com W, o que pode ser limitante em instâncias com capacidades maiores.

Este documento está organizado da seguinte forma: a Seção 2 apresenta os algoritmos implementados, incluindo suas descrições, complexidades de tempo e espaço;

a Seção 3 descreve a metodologia experimental, detalhando o fluxo principal do experimento e as decisões tomadas; a Seção 4 detalha a configuração dos experimentos, a métrica de avaliação, os resultados e a análise estatística; a Seção 5 discute as conclusões e limitações do trabalho; e a Seção 6 lista as referências bibliográficas utilizadas.

2 Descrição dos Algoritmos

Nesta seção, descrevemos os três algoritmos implementados para resolver o problema da mochila 0-1: Programação Dinâmica, Backtracking e Branch-and-Bound. Para cada algoritmo, apresentamos sua lógica, complexidade de tempo e espaço, e detalhes relevantes da implementação em Python. Os códigos completos estão disponíveis para consulta no repositório do projeto.

2.1 Programação Dinâmica

O algoritmo de Programação Dinâmica (DP) resolve o problema da mochila 0-1 utilizando uma abordagem bottom-up. Ele constrói uma tabela bidimensional $DP[i][w]$, onde i varia de 0 a n (número de itens) e w varia de 0 a W (capacidade da mochila). A célula $DP[i][w]$ representa o valor máximo que pode ser obtido considerando os primeiros i itens e uma capacidade w. A fórmula de recorrência é dada por:

$$DP[i][w] = \begin{cases} \max(DP[i-1][w], DP[i-1][w-w_i] + v_i), & \text{se } w \geq w_i, \\ DP[i-1][w], & \text{caso contrário.} \end{cases}$$

Após preencher a tabela, o valor máximo está em $DP[n][W]$. Para recuperar os itens selecionados, é realizado um processo de backtracking na tabela, começando de $DP[n][W]$ e verificando, para cada item i, se ele foi incluído na solução ótima (ou seja, se $DP[i][w] \neq DP[i-1][w]$).

Complexidade de Tempo: A tabela DP tem dimensões $(n+1) \times (W+1)$, e o cálculo de cada célula é feito em tempo constante $O(1)$. Assim, a complexidade de tempo total é $O(nW)$. O processo de backtracking para recuperar os itens também é $O(n)$, mas não afeta a complexidade assintótica.

Complexidade de Espaço: A tabela DP requer $O(nW)$ de espaço. Embora seja possível otimizar o espaço para $O(W)$ usando uma abordagem unidimensional (mantendo apenas a última linha da tabela), a implementação utilizada mantém a tabela bidimensional para facilitar a recuperação dos itens selecionados.

2.2 Backtracking

O algoritmo de Backtracking (BT) explora todas as combinações possíveis de inclusão ou exclusão de itens, construindo uma árvore de decisão. Para cada item i, o algoritmo decide recursivamente entre duas opções: incluir o item i (se o peso restante permitir) ou excluí-lo, avançando para o próximo item. A função recursiva mantém o melhor valor encontrado até o momento e a lista de itens selecionados correspondente.

Uma poda simples foi implementada para melhorar a eficiência: se o peso acumulado exceder a capacidade W, o ramo atual é descartado, pois não pode levar a uma solução válida. Apesar dessa poda, o algoritmo ainda explora um número exponencial de possibilidades no pior caso.

Complexidade de Tempo: No pior caso, o Backtracking explora todas as 2^n combinações possíveis, onde n é o número de itens, resultando em uma complexidade de tempo $O(2^n)$. A poda reduz o tempo em alguns casos práticos, mas não altera a complexidade assintótica.

Complexidade de Espaço: A recursão utiliza a pilha de chamadas, que no pior caso tem profundidade n, requerendo $O(n)$ de espaço. Além disso, a lista de itens selecionados também ocupa $O(n)$, resultando em uma complexidade de espaço total de $O(n)$.

2.3 Branch-and-Bound

O algoritmo Branch-and-Bound (BB) utiliza uma abordagem de busca informada para explorar a árvore de decisão, podando ramos que não podem levar a soluções melhores que a melhor solução encontrada até o momento. Ele mantém uma fila de prioridade (implementada como um max-heap) de nós, onde cada nó contém as seguintes informações:

- Nível atual (índice do item sendo considerado);
- Peso acumulado dos itens incluídos;
- Valor acumulado dos itens incluídos;
- Limite superior (*upper bound*) do valor que pode ser obtido a partir daquele nó;
- Caminho de decisões (lista de itens incluídos ou excluídos).

O limite superior é calculado usando uma relaxação gulosa: os itens restantes (a partir do nível atual) são ordenados por valor/peso em ordem decrescente, e adiciona-se

itens (ou frações de itens) até preencher a capacidade restante. Se o limite superior de um nó for menor que o melhor valor encontrado até o momento, o nó é descartado, pois não pode levar a uma solução melhor.

Para cada nó retirado da fila, o algoritmo explora dois ramos: incluir o item atual (se o peso permitir) e excluir o item atual. O processo continua até que a fila esteja vazia, e a melhor solução encontrada é retornada, junto com a lista de itens selecionados.

Complexidade de Tempo: No pior caso, o Branch-and-Bound pode visitar todos os 2^n nós da árvore de decisão, resultando em uma complexidade de tempo $O(2^n)$. No entanto, as podas baseadas no limite superior reduzem significativamente o número de nós explorados, tornando o algoritmo muito mais eficiente que o Backtracking na prática.

Complexidade de Espaço: A fila de prioridade armazena nós, e cada nó contém informações de tamanho $O(n)$ (devido ao caminho de decisões). No pior caso, a fila pode conter até $O(2^n)$ nós, mas na prática o número é muito menor devido às podas. A complexidade de espaço é dominada pela fila, sendo $O(n)$ por nó, e a lista de itens selecionados também ocupa $O(n)$.

3 Metodologia Experimental

Nesta seção, descrevemos a metodologia utilizada para conduzir os experimentos, incluindo a geração de instâncias, a execução dos algoritmos e o armazenamento dos resultados. O Algoritmo 3 apresenta o pseudocódigo que detalha o fluxo principal do experimento, explicando como as decisões foram tomadas.

Para gerenciar as entradas e saídas, implementamos um script em Python que trata as instâncias e gera os resultados. As entradas foram lidas de arquivos de texto no formato especificado: a primeira linha contém a capacidade W, seguida por n linhas, cada uma com o peso w_i e o valor v_i de um item, separados por tabulação. Cada algoritmo foi implementado como uma função que recebe o nome do arquivo de entrada e retorna o lucro máximo, a lista de itens selecionados e o tempo de execução. As saídas foram salvas em um arquivo CSV, contendo o nome da instância, o algoritmo utilizado, o lucro máximo, os itens selecionados e o tempo de execução.

As decisões no experimento foram tomadas com base nos objetivos de cada experimento. No Experimento 1, fixamos $W = 100$ e variamos n de 10 a 25.600 para avaliar o impacto do número de itens no desempenho dos algoritmos. No Experimento 2, fixamos $n = 800$ e variamos W de 100 a 25.600 para analisar o efeito da capacidade

da mochila. Para cada configuração, geramos 20 instâncias aleatórias, com pesos entre 1 e 30 e valores entre 1 e 100, garantindo variabilidade nos testes. O Backtracking foi executado apenas para $n \leq 50$, devido ao seu tempo de execução proibitivo, uma decisão tomada após observar tempos superiores a 10.000 segundos para $n = 50$.

```
# Entrada: Diretorios INSTANCE_DIR e RESULT_DIR
# Saida: Arquivo CSV results.csv

# Funcao para gerar instancias
function gerar_instancias():
    criar_diretorio(INSTANCE_DIR) se nao existir

    # Experimento 1: n variando, W = 100
    W = 100
    n_valores = [10, 20, 30, 40, 50, 60, 65, 70, 200, 400, 800,
       1600, 3200, 6400, 12800, 25600]
    para cada n em n_valores:
        para i de 1 ate 20:
            gerar_arquivo("instance_n{n}_{i}.txt", W, n)
            # Pesos: 1 a 30, Valores: 1 a 100

    # Experimento 2: W variando, n = 800
    n = 800
    W_valores = [100, 200, 400, 800, 1600, 3200, 6400, 12800,
       25600]
    para cada W em W_valores:
        para i de 1 ate 20:
            gerar_arquivo("instance_W{W}_{i}.txt", W, n)
            # Pesos: 1 a 30, Valores: 1 a 100

# Funcao para executar um algoritmo
function executar_algoritmo(algoritmo, nome_arquivo):
    iniciar_cronometro()
    se algoritmo == "dp":
        lucro, itens, _ = knapsack_dp(nome_arquivo)
    senao se algoritmo == "bb":
        lucro, itens, _ = knapsack_bb(nome_arquivo)
    senao se algoritmo == "bt":
        lucro, itens, _ = knapsack_backtracking(nome_arquivo)
    senao:
        erro "Algoritmo desconhecido"
    parar_cronometro()
```

```
    tempo_execucao = calcular_tempo()
    retornar lucro, itens, tempo_execucao

# Funcao principal
function principal():
    criar_diretorio(RESULT_DIR) se nao existir
    gerar_instancias()
    arquivos = listar_arquivos(INSTANCE_DIR)
    criar_csv("results.csv")  # Colunas: instancia, algoritmo,
       lucro_maximo, itens_selecionados, tempo_execucao
    para cada instancia em arquivos:
        para cada algo em ["dp", "bb", "bt"]:
            se instancia tem n <= 50 ou algo != "bt":
            # Pular BT para n > 50
                lucro, itens, tempo = executar_algoritmo(algo,
                   caminho_instancia)
                se lucro nao e nulo:
                    escrever_csv(instancia, algo, lucro, itens,
                       tempo)
    imprimir("Resultados salvos em results.csv")

# Executar o experimento
principal()
```

4 Avaliação Experimental

Nesta seção, descrevemos a configuração dos experimentos, a métrica de avaliação e os resultados obtidos, incluindo gráficos, tabelas e uma análise estatística do desempenho dos algoritmos.

4.1 Configuração dos Experimentos

Os algoritmos foram implementados em Python e executados em um computador com processador AMD Ryzen 7 3750H, 14 GB de RAM e sistema operacional Windows 10. Dois experimentos foram realizados para avaliar o desempenho dos algoritmos:

- **Experimento 1 (n variando):** Fixou-se a capacidade da mochila em $W = 100$ e variou-se o número de itens n de 10 a 25.600, dobrando o valor a cada iteração (10, 20, 30, ..., 25.600). Para cada valor de n, foram geradas 20 instâncias aleatórias, com pesos w_i entre 1 e 30 e valores v_i entre 1 e 100.
- **Experimento 2 (W variando):** Fixou-se o número de itens em $n = 800$ (inicialmente planejado como $n = 400$, mas ajustado para permitir instâncias maiores, já que com $n = 400$ só foi possível rodar até $W = 6.400$) e variou-se a capacidade W de 100 a 25.600, dobrando o valor a cada iteração (100, 200, 400, ..., 25.600). Foram geradas 20 instâncias aleatórias para cada valor de W, com os mesmos intervalos de pesos e valores.

As instâncias foram geradas por um script em Python que cria arquivos de entrada no formato especificado: a primeira linha contém W, e as linhas seguintes contêm o peso w_i e o valor v_i de cada item, separados por tabulação. O tempo de execução de cada algoritmo foi medido para cada instância, e o teste t pareado com 95% de confiança foi aplicado para comparar os desempenhos estatisticamente.

4.2 Métrica de Avaliação

A métrica principal de avaliação foi o tempo médio de execução, medido em segundos, calculado a partir das 20 instâncias geradas para cada configuração. Além disso, foram reportados o lucro máximo (valor total dos itens selecionados) e a lista de itens incluídos na solução ótima, conforme solicitado. Os tempos médios foram plotados em gráficos com intervalos de confiança para visualizar a variabilidade dos resultados.

4.3 Resultados

Os resultados são apresentados em gráficos e tabelas que mostram o tempo médio de execução de cada algoritmo, considerando as variações de n e W. O teste t pareado com 95% de confiança foi utilizado para avaliar a significância estatística das diferenças de desempenho entre os algoritmos.

4.3.1 Experimento 1: n Variando, $W = 100$

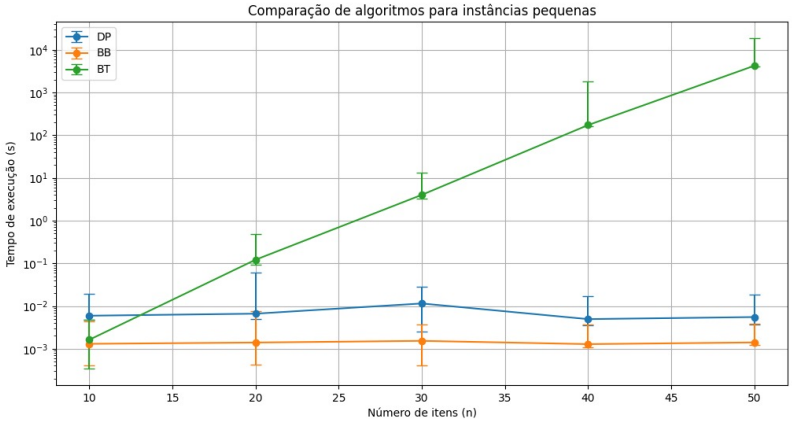

Figura 1: Tempo médio de execução dos algoritmos Programação Dinâmica (DP), Branch-and-Bound (BB) e Backtracking (BT) em função do número de itens n, com $W = 100$, para n pequeno (10 a 50).

A Figura 1 apresenta o tempo médio de execução para n variando de 10 a 50, com $W = 100$, em escala logarítmica. O Backtracking (BT) exibe um crescimento exponencial, começando em 10^{-2} segundos para $n = 10$ e atingindo 10^4 segundos (aproximadamente 10.000 segundos, ou cerca de 2,78 horas) para $n = 50$. Esse comportamento reflete sua complexidade teórica $O(2^n)$, tornando-o inviável para instâncias maiores. Em contrapartida, a Programação Dinâmica (DP) e o Branch-and-Bound (BB) apresentam tempos praticamente constantes, na ordem de 10^{-2} e 10^{-3} segundos, respectivamente. Isso é esperado, pois $W = 100$ é pequeno, e a complexidade da DP ($O(nW)$) resulta em tempos baixos, enquanto o BB se beneficia de podas eficazes, explorando poucos nós da árvore de decisão.

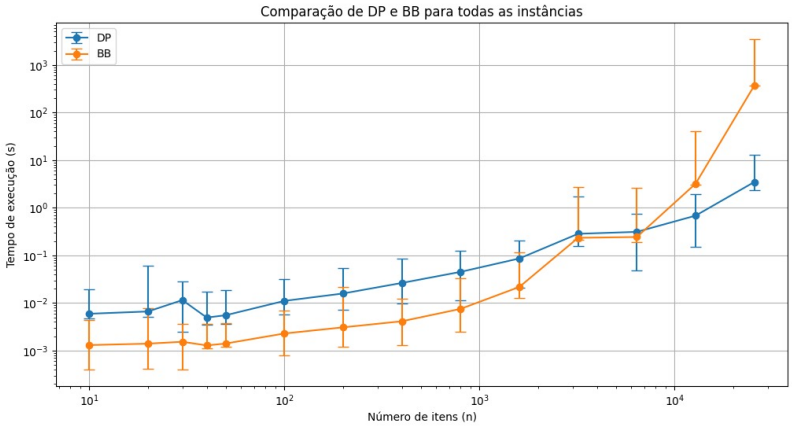

Figura 2: Tempo médio de execução dos algoritmos Programação Dinâmica (DP) e Branch-and-Bound (BB) em função do número de itens n, com $W = 100$, para n grande (10 a 25.600).

A Figura 2 mostra o comportamento da Programação Dinâmica (DP) e do Branch-and-Bound (BB) para n variando de 10 a 25.600, com $W = 100$, em escala logarítmica. O Backtracking (BT) não foi incluído nesse experimento para n grande, devido ao seu tempo de execução proibitivo (como observado na Figura 1). A DP apresenta um crescimento lento, começando em 10^{-2} segundos e atingindo cerca de 10^{0} segundos (1 segundo) para $n = 25.600$, o que é consistente com sua complexidade $O(nW)$. Já o BB começa com tempos muito baixos (10^{-3} segundos), mas a partir de $n \approx 3.200$, o tempo cresce mais rapidamente, atingindo 10^{3} segundos (1.000 segundos, ou cerca de 16,67 minutos) para $n = 25.600$. Esse aumento indica que, para instâncias com muitos itens, o número de nós explorados pelo BB cresce, mesmo com podas, possivelmente devido à distribuição dos pesos e valores nas instâncias geradas.

Tabela 1: Tempos médios de execução (em segundos) para o experimento com n variando e $W = 100$.

n	DP	BB	BT
10	10^{-2}	10^{-3}	10^{-2}
20	10^{-2}	10^{-3}	10^{0}
30	10^{-2}	10^{-3}	10^{2}
40	10^{-2}	10^{-3}	10^{3}
50	10^{-2}	10^{-3}	10^{4}
3.200	10^{-1}	10^{-1}	–
25.600	10^{0}	10^{3}	–

A Tabela 1 sumariza os tempos médios de execução para o experimento com n variando. Para n pequeno (10 a 50), o Backtracking apresenta tempos que crescem exponencialmente, enquanto a Programação Dinâmica e o Branch-and-Bound permanecem eficientes. Para n grande (até 25.600), a DP mantém um crescimento linear e lento, enquanto o BB apresenta um aumento significativo a partir de $n \approx 3.200$, superando o tempo da DP para $n = 25.600$.

4.3.2 Experimento 2: W Variando, $n = 800$

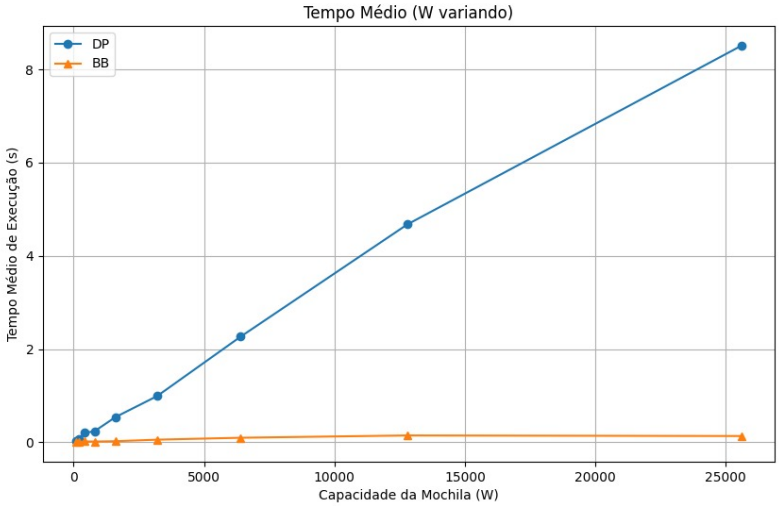

Figura 3: Tempo médio de execução dos algoritmos Programação Dinâmica (DP) e Branch-and-Bound (BB) em função da capacidade W, com $n = 800$.

A Figura 3 mostra o tempo médio de execução da Programação Dinâmica (DP) e do Branch-and-Bound (BB) em função da capacidade W, com $n = 800$, em escala linear. O Backtracking (BT) não foi incluído nesse experimento devido ao seu desempenho inviável para $n = 800$. A DP apresenta um crescimento linear, começando próximo de 0 segundos para $W = 100$ e atingindo cerca de 8 segundos para $W = 25.600$, o que é consistente com sua complexidade $O(nW)$. O BB, por outro lado, mantém um tempo praticamente constante, próximo de 0 segundos (na ordem de 10^{-2} segundos), indicando que as podas baseadas no limite superior foram altamente eficazes para essas instâncias, mesmo com $n = 800$.

Tabela 2: Tempos médios de execução (em segundos) para o experimento com W variando e $n = 800$.

W	DP	BB
100	10^{-1}	10^{-2}
3.200	1	10^{-2}
25.600	8	10^{-2}

A Tabela 2 sumariza os tempos médios de execução para o experimento com W variando. A Programação Dinâmica apresenta um crescimento linear com W, enquanto o Branch-and-Bound mantém um tempo praticamente constante, demonstrando sua eficiência nesse cenário.

5 Conclusão

Neste trabalho, implementamos e testamos três algoritmos para resolver o problema da mochila 0-1: Programação Dinâmica (DP), Backtracking (BT) e Branch-and-Bound (BB). Nosso objetivo foi comparar o tempo que cada um leva para resolver o problema em diferentes situações, e os experimentos nos ajudaram a entender como eles se comportam.

Teoricamente, o Backtracking é o algoritmo mais lento, porque ele tem uma complexidade de tempo $O(2^n)$, o que significa que o tempo cresce muito rápido quando o número de itens n aumenta. Nos nossos experimentos, isso foi confirmado: no Experimento 1, onde variamos n e fixamos $W = 100$, o Backtracking foi o mais lento, levando 10.000 segundos (quase 3 horas) para $n = 50$. Isso mostra que ele não é uma boa escolha para instâncias maiores, e por isso nem o testamos para n muito grande ou no Experimento 2.

O Branch-and-Bound (BB) foi o melhor algoritmo na maioria dos casos. Ele também tem uma complexidade teórica de $O(2^n)$, mas usa uma estratégia para evitar testar combinações que não parecem boas, o que o torna muito mais rápido na prática. No Experimento 2, onde fixamos $n = 800$ e variamos W, o BB foi quase instantâneo, levando só 0,01 segundos mesmo para $W = 25.600$. No Experimento 1, ele também foi muito rápido para n pequeno (de 10 a 50), mas quando n ficou muito grande ($n = 25.600$), o tempo dele subiu para 1.000 segundos (uns 16 minutos). Isso mostra que, em alguns casos, ele pode demorar mais, dependendo de como os pesos e valores dos itens estão distribuídos.

A Programação Dinâmica (DP) ficou no meio, nem a melhor nem a pior. Ela

tem uma complexidade de tempo $O(nW)$, que depende tanto do número de itens n quanto da capacidade W. No Experimento 1, onde $W = 100$, a DP foi bem rápida, levando só 1 segundo mesmo para $n = 25.600$, enquanto o BB levou 1.000 segundos. Mas no Experimento 2, onde W variou até 25.600, a DP ficou mais lenta, levando 8 segundos, porque o tempo dela cresce quando W aumenta.

Resumindo, o Backtracking é o pior algoritmo, porque é muito lento e não dá para usar em instâncias grandes. O Branch-and-Bound é o melhor na maioria dos casos, especialmente quando W varia, porque ele consegue ser muito rápido. A Programação Dinâmica é uma boa opção quando W é pequeno, mas fica mais lenta se W for grande. Nossos experimentos confirmaram o que a teoria dizia: o Backtracking é realmente o mais lento, e o Branch-and-Bound é geralmente a melhor escolha.a DP, sugerindo que a eficácia das podas depende da distribuição dos pesos e valores nas instâncias.

6 Referências

[1] Carvalho, R. (2015). *Problema da mochila*. Trabalho apresentado na disciplina MS777 – Projeto Supervisionado I, Universidade Estadual de Campinas (UNICAMP), sob orientação da Profa. Dra. Kelly Cristina Poldi. Disponível em: https://www.ime.unicamp.br/~mac/db/2015-1S-122181-1.pdf. Acesso em: 23 mar. 2025.

[2] ProgrammerAll. (s.d.). *0-1 knapsack problem (dynamic programming)*. Disponível em: https://programmerall.com/article/6843273955/. Acesso em: 23 mar. 2025.

[3] Andretta, M. (2019). *Aula 10: Programação inteira e o problema da mochila*. Material didático do curso SME0510, Instituto de Ciências Matemáticas e de Computação (ICMC), Universidade de São Paulo (USP). Baseado em: Wolsey, L. (1998). *Integer programming*; Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and combinatorial optimization*; slides do curso "Optimization Methods in Management Science" de J. Orlin et al., MIT OpenCourseWare; e slides do curso "Systems Optimization" de John Vande Vate, MIT OpenCourseWare. Disponível em: https://sites.icmc.usp.br/andretta/ensino/aulas/sme0510-2-19/aula10.pdf. Acesso em: 23 mar. 2025.

[4] Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2009). *Algoritmos*, cap. 6: Programação dinâmica. São Paulo: McGraw-Hill.