Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Avaliação Empírica de Algoritmos para o Problema da Mochila 0-1: Programação Dinâmica, Backtracking e Branch-and-Bound

BCC241 - Projeto e Análise de Algoritmos

Bárbara Rodrigues Mateus, Caio Lucas Pereira da Silva, Gustavo Zacarias Souza

Professor: Anderson Almeida Ferreira

Ouro Preto
11 de agosto de 2025

Sumário

Lista de Figuras

Lista de Tabelas

1 Resumo

Neste trabalho, exploramos o famoso problema da mochila 0-1. É um problema muito conhecido e com diversas aplicações práticas, como a divisão de um montante entre várias opções de investimento. Devido à sua importância no campo teórico e por ser um problema NP-difícil, muitos pesquisadores têm se dedicado a ele, procurando criar métodos de resolução mais eficazes que aproveitem as particularidades de cada versão.

Neste estudo, usaremos três métodos tradicionais de resolução: programação dinâmica, retrocesso e branch-and-bound. A avaliação será feita de maneira prática, comparando o desempenho de cada método por meio de testes controlados, com a criação automática de exemplos e a análise dos tempos de execução e da qualidade das soluções obtidas.

2 Introdução

2.1 Problema

Suponhamos um empreendedor que é dono de um pequeno food truck e vai participar de um festival gastronômico. Ele dispõe de um espaço limitado dentro do veículo para transportar ingredientes e equipamentos, mas deseja levar aqueles que possibilitem preparar os pratos mais lucrativos durante o evento.

Ele faz uma lista com todos os itens que poderia levar: sacos de farinha, caixas de legumes, carne, temperos, bebidas, utensílios e até uma chapa extra. Cada item possui um peso (ou volume) e um valor estimado de retorno financeiro, baseado no quanto pode contribuir para suas vendas.

O problema surge porque o espaço dentro do food truck é limitado — por exemplo, se o veículo comporta até 200 kg, não será possível levar tudo. Isso gera a seguinte pergunta: como escolher quais itens levar para maximizar o lucro total?

Uma solução ingênua seria adotar o “método guloso”: pegar o item de maior valor de retorno e colocá-lo primeiro, depois o segundo de maior valor que caiba no espaço restante, e assim por diante. Porém, essa abordagem pode levar a escolhas ruins: pode acabar ocupando grande parte do espaço com um item pesado, deixando de fora vários outros mais leves que, juntos, gerariam maior lucro.

Este é o chamado problema da mochila: dado um conjunto de itens, cada um com peso e valor, e uma capacidade máxima do recipiente (no caso, o espaço ou peso suportado pelo food truck), escolher um subconjunto de itens que maximize o valor total, sem ultrapassar a capacidade.

Esse problema é amplamente estudado em ciência da computação, pois modela situações de escolha ótima sob restrição de recursos e pertence à classe dos problemas NP-hard.

2.2 Objetivo do Trabalho

O objetivo deste trabalho é desenvolver, implementar e avaliar empiricamente três algoritmos para a resolução do problema da Mochila 0-1 sem repetição: programação

dinâmica, backtracking e branch-and-bound. Esse problema é um clássico da Computação e pertence à classe dos problemas NP-completos, sendo também NP-difícil na sua formulação de otimização. Isso significa que, para grandes instâncias, não se conhece nenhum algoritmo determinístico de tempo polinomial que garanta a solução ótima, o que torna essencial a análise de diferentes estratégias de resolução.

A proposta envolve analisar o desempenho dessas abordagens em diferentes tamanhos de instâncias, medindo e comparando seus tempos de execução. Para isso, serão geradas instâncias aleatórias com variação no número de itens (n) e na capacidade máxima da mochila (W), de forma a verificar o impacto do crescimento do problema no tempo de execução.

Cada experimento será repetido múltiplas vezes para permitir análise estatística, incluindo teste t pareado com 95

Por fim, pretende-se apresentar e interpretar os resultados por meio de gráficos com intervalos de confiança, destacando as diferenças de eficiência entre os métodos e discutindo suas limitações frente ao crescimento exponencial do espaço de busca inerente a problemas NP-difíceis.

2.3 Resultados obtidos

2.3.1 Experimento 1

Nesse experimento, apenas o número de itens n variou, enquanto a capacidade da mochila (W) foi mantida em um valor fixo. Dito isso, o Branch and Bound apresentou tempos irregulares, pois seu desempenho não depende somente de n, mas também de como os pesos e os valores dos itens encontram-se dispostos na entrada, influenciando na eficiência da poda. Enquanto isso, o Backtracking teve um crescimento mais

2.3.2 Experimento 2

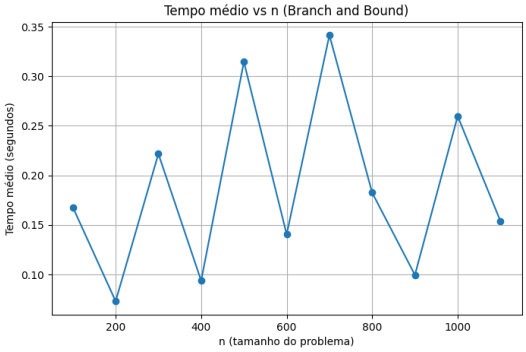

Figura 1: Resultados do Branch and Bound

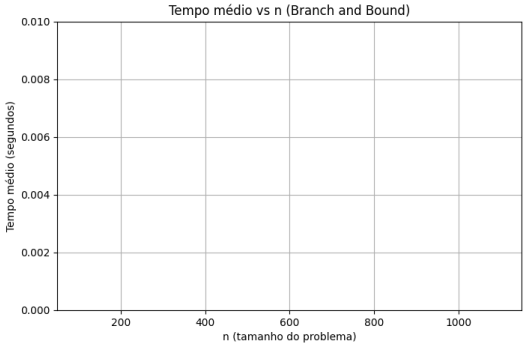

Figura 2: Resultados do Branch and Bound (Escala normalizada)

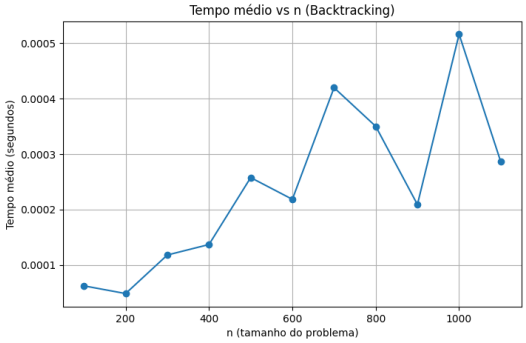

Figura 3: Resultados do Backtracking

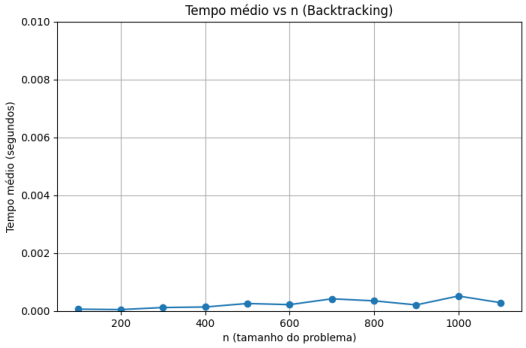

Figura 4: Resultados do Backtracking (Escala normalizada)

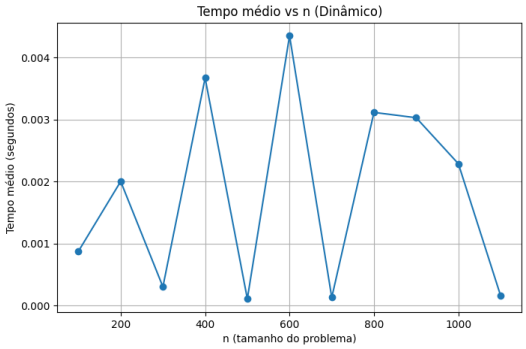

Figura 5: Resultados da Prog. Dinâmica

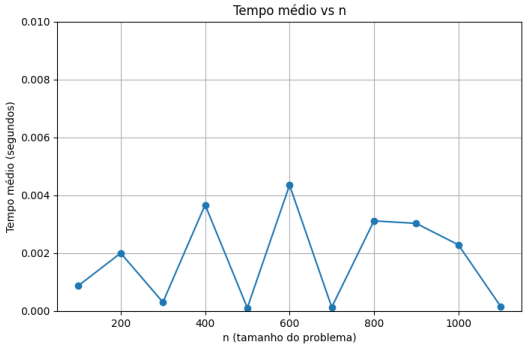

Figura 6: Resultados da Prog. Dinâmica (Escala normalizada)

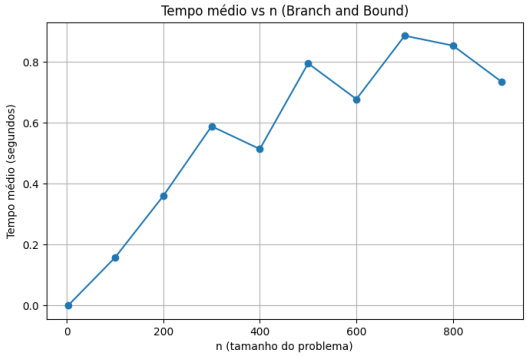

Figura 7: Resultados do Branch and Bound

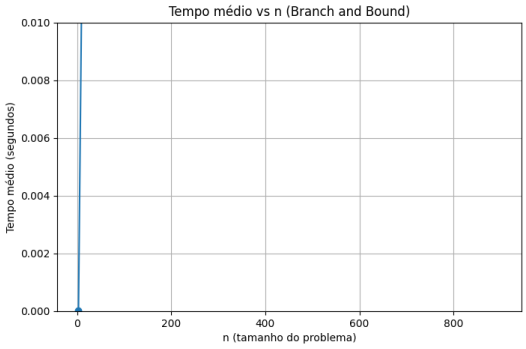

Figura 8: Resultados do Branch and Bound (Escala normalizada)

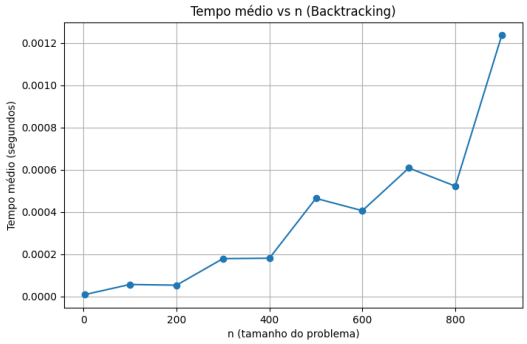

Figura 9: Resultados do Backtracking

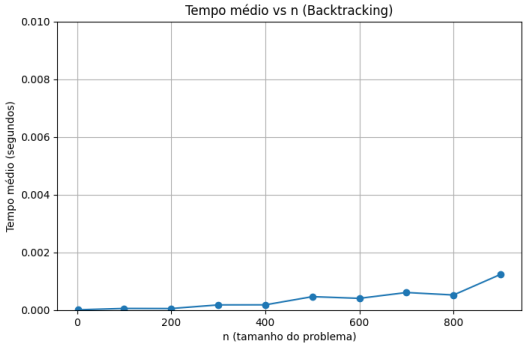

Figura 10: Resultados do Backtracking (Escala normalizada)

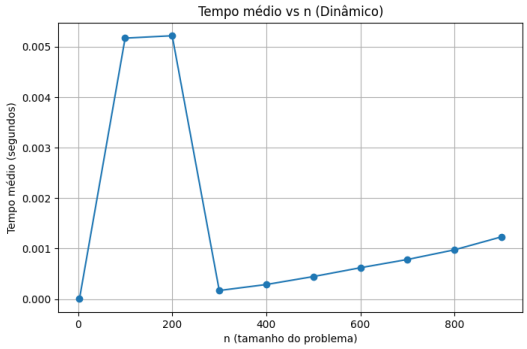

Figura 11: Resultados da Prog. Dinâmica

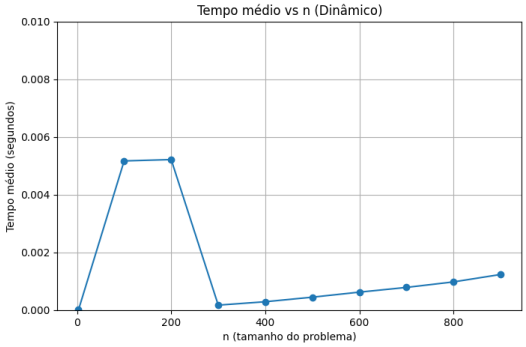

Figura 12: Resultados da Prog. Dinâmica (Escala normalizada)

3 Descrição dos algoritmos e análise de complexidade

3.1 Branch and Bound

A ideia do algoritmo é representar todas as combinações possíveis de itens como uma **árvore de decisão**, onde cada nó indica se um item foi incluído ou não. Para reduzir o espaço de busca, é utilizada a técnica de *branch and bound*:

1. **Ordenação dos itens**
   Os itens são ordenados pela razão valor/peso em ordem decrescente (estratégia gulosa para melhorar o cálculo dos limites).

2. **Nó raiz**
   Cria-se um nó inicial (nível -1, lucro e peso zero) e insere-o em uma fila para busca em largura (BFS).

3. **Expansão dos nós**
   Enquanto a fila não estiver vazia:

   - Remove-se um nó u da fila.
   - Cria-se o nó `include` (incluindo o próximo item) e calcula-se:
     - Novo peso.
     - Novo lucro.
     - Limite superior (*bound*) usando uma solução fracionária gulosa.
     - Se o peso for $\leq$ capacidade e o lucro for melhor, atualiza o melhor lucro.
     - Se o limite (*bound*) for maior que o melhor lucro atual, mantém o nó para expansão futura.
   - Cria-se o nó `exclude` (excluindo o próximo item) e calcula-se:
     - Limite (*bound*).
     - Se o limite for promissor, mantém o nó para expansão futura.

4. **Reconstrução da solução**
   Depois de processar todos os nós, volta-se pelo caminho do melhor nó encontrado (`bestNode`) para determinar quais itens foram escolhidos.

5. **Limpeza de memória**
   Todos os nós alocados são liberados para evitar vazamento de memória.

3.1.1 Complexidade de Tempo

- **Pior caso:** $O(2^n)$
  Mesmo com podas, no pior cenário é preciso visitar praticamente todos os subconjuntos possíveis de itens.

- **Melhor caso:** $O(n \log n)$
  Quando a poda é muito eficiente e poucos nós são explorados (apenas ordenação inicial e poucas expansões).
- **Caso médio:** Entre $O(2^n)$ e $O(n \log n)$, dependendo da eficiência da poda para os dados de entrada.

O cálculo do *bound* para cada nó é $O(n)$, e pode ocorrer em até $O(2^n)$ nós, mas normalmente o número real de nós é muito menor devido à poda.

3.1.2 Complexidade de Espaço

- **Memória para itens:** $O(n)$
- **Memória para nós:** No pior caso, pode chegar a $O(2^n)$ se quase todos os nós forem mantidos na fila.
- **Vetor** `takenItems`**:** $O(n)$
- **Fila de BFS:** Até $O(2^n)$ nós no pior caso, mas tipicamente bem menor.

3.2 Backtracking Iterativo com Poda por Limite Superior

O objetivo do algoritmo é explorar todas as combinações possíveis de inclusão ou exclusão de itens, evitando caminhos que não podem gerar uma solução melhor do que a já encontrada. A técnica foi implementada com **poda por limite superior**, implementado de forma iterativa usando uma pilha para simular a busca em profundidade (DFS).

1. **Pré-processamento**
   Os itens são ordenados pela razão valor/peso em ordem decrescente, para maximizar a eficiência da poda baseada em limite superior.
2. **Função de limite superior (*upper bound*)**
   Dado um estado $(i, \text{peso atual}, \text{valor atual})$, calcula o melhor valor possível que pode ser obtido completando a mochila de forma gulosa (adicionando itens inteiros e fracionados, se necessário).
3. **Inicialização**
   Uma pilha é criada para armazenar o estado da busca: índice do item atual, peso acumulado, valor acumulado e vetor de seleção de itens.
4. **Exploração iterativa**
   Enquanto a pilha não estiver vazia:
   - Retira-se um estado do topo.
   - **Caso base:** se todos os itens foram considerados, atualiza-se o lucro máximo se necessário.

- **Poda:** se o limite superior desse estado for menor ou igual ao lucro máximo atual, o caminho é descartado.
- **Ramos:**
  (a) **Excluir** o item atual: adiciona à pilha o estado sem aumentar peso nem valor.
  (b) **Incluir** o item atual (se couber na mochila): adiciona à pilha o estado com peso e valor incrementados, marcando o item como escolhido.

5. **Reconstrução da solução**
   Ao final, o vetor de itens escolhidos é reordenado para corresponder à ordem original de entrada.

3.2.1 Complexidade de Tempo

- **Pior caso:** $O(2^n)$
  O algoritmo pode explorar todas as combinações possíveis de itens.
- **Melhor caso:** $O(n \log n)$
  Quando a poda elimina quase todos os ramos logo no início (apenas a ordenação inicial e poucas expansões são necessárias).
- **Caso médio:** Entre $O(2^n)$ e $O(n \log n)$, dependendo da eficiência da poda para o conjunto de entrada.

O cálculo do limite superior é $O(n)$ e pode ser realizado para cada estado explorado.

3.2.2 Complexidade de Espaço

- **Memória para itens:** $O(n)$
- **Memória para vetor de seleção:** $O(n)$ por estado na pilha.
- **Memória para pilha:** no pior caso, $O(2^n)$ estados, mas tipicamente muito menor devido à poda.

3.3 Programação Dinâmica

3.3.1 Descrição do algoritmo

A abordagem utilizada é **bottom-up**, que preenche uma matriz dp onde $dp[i][w]$ representa o lucro máximo que pode ser obtido usando os primeiros i itens com capacidade w. Além disso, o algoritmo armazena soluções parciais em uma tabela bidimensional para evitar recomputações.

1. **Inicialização**
   Criar uma matriz dp de dimensões $(n + 1) \times (W + 1)$ inicializada com zeros, onde n é o número de itens e W é a capacidade da mochila.

2. **Preenchimento da tabela**
   Para cada item i (de 1 a n) e para cada capacidade w (de 1 a W):

   - Se o peso do item i é menor ou igual a w, escolher o melhor entre:
     - Não incluir o item: $dp[i-1][w]$
     - Incluir o item: valor do item $+$ melhor valor para a capacidade restante $dp[i-1][w-peso]$
   - Caso contrário, herdar o valor de $dp[i-1][w]$.

3. **Recuperação da solução**
   O valor máximo está em $dp[n][W]$. Para reconstruir os itens escolhidos, percorre-se a tabela de baixo para cima verificando quais decisões resultaram na inclusão de cada item.

4. **Armazenamento do resultado**
   O vetor `takenItems` é atualizado para marcar quais itens foram selecionados.

3.3.2 Complexidade de Tempo

O preenchimento da tabela requer $O(n \times W)$ operações, pois cada célula é calculada em tempo constante.

3.3.3 Complexidade de Espaço

- A matriz dp ocupa $O(n \times W)$ posições.
- O vetor de seleção `takenItems` ocupa $O(n)$.

4 Avaliação experimental

- Experimento 1:
  - Objetivo: medir o tempo de execução dos algoritmos variando somente o números de itens (n), mantendo a capacidade da mochila (W) fixa.
  - Instâncias: Foram geradas instâncias com n variando de 100 a 1100, incrementado em passos de 100. Para cada valor de n, foram geradas 10 instâncias independentes. Os pesos dos itens foram sorteados de forma uniforme no intervalo [1, n/4] e os valores no intervalo [1, 1000]. As instâncias do primeiro experimento foram salvas em arquivos separados na pasta `instâncias`, com nomes no formato `instancia_n{n}{i}.txt`.
  - Configuração inicial: começa com n = 100 e W = 100.
  - Procedimento: A cada iteração n aumenta em 100, mas W permanece constante. Exemplos de valores:
    - Iteração 1: n = 100, W = 100
    - Iteração 2: n = 200, W = 100

    - Iteração 3: n = 300, W = 100

- Experimento 2:
  - Objetivo: medir o tempo de execução variando ambos o número de itens n e a capacidade da mochila W.
  - Instâncias: Foram geradas instâncias com n variando de 100 a 900, também em passos de 100, com 10 instâncias para cada configuração. Enquanto a capacidade da mochila (W) variou simultaneamente a n, assumindo o mesmo valor de n a cada instância. Os pesos dos itens foram sorteados de forma uniforme no intervalo [1, n/4] e os valores no intervalo [1, 1000].
  - Configuração inicial: começa com n = 100 e W = 100.
  - Procedimento: A cada iteração n e W aumentam em 100. Exemplos de valores:
    - Iteração 1: n = 100, W = 100
    - Iteração 2: n = 200, W = 200
    - Iteração 3: n = 300, W = 300

4.1 Teste estatístico

Para realizar os testes estatísticos requeridos no trabalho, foi criado um arquivo .cpp afim de realizar os cálculos do teste t onde $t = \frac{\bar{x}_d}{\frac{s_d}{\sqrt{n}}}$. Os resultados obtidos foram os seguintes:

Algoritmo	**Média (s)**	**Desvio Padrão (s)**	**Erro Padrão (s)**	**Número de Instâncias**
Branch And Bound	0.612199	0.244367	0.025617	91
Dynamic	0.001638	0.006623	0.000694	91
Backtracking	0.000408	0.001069	0.000112	91

Tabela 1: Resultados estatísticos por algoritmo

A partir desta tabela, as seguintes comparações foram realizadas:
Comparação entre algoritmos

Comparações	Estatística t	Valor p	Intervalo Confiança (±s)	Rejeita H0	Conclusão
Branch And Bound vs Dynamic	23.7394	0.001	0.050410	SIM	Algoritmo 2 é estatisticamente melhor (p ¡ 0.05)
Branch And Bound vs Backtracking	23.9043	0.001	0.050163	SIM	Algoritmo 2 é estatisticamente melhor (p ¡ 0.05)
Dynamic vs Backtracking	1.7385	0.100	0.001387	NÃO	EMPATE ESTATÍSTICO - Não há diferença significativa (p $\geq$ 0.05)

Tabela 2: Comparações estatísticas entre algoritmos

Algoritmo	Complexidade de tempo	Complexidade de espaço	Uso prático
Branch & Bound	$O(2^n)$	$O(2^n)$	Bom para $n < 100$
Programação Dinâmica	$O(nW)$	$O(nW)$	Bom para W pequeno
Backtracking	$O(2^n)$	$O(n)$	Parecido com o B&B

Tabela 3: Comparação dos algorítmos por complexidade e perfomance.

5 Referências Bibliográficas

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[2] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 1st edition, 2005.

[3] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[5] S. He and Q. Gu. A new hybrid algorithm for the 0-1 knapsack problem. *2011 International Conference on Signal Processing, Communication, Computing and Mechatronics (SPCCM)*, pages 729–732, 2011.

[6] Bernard M.E. Moret. On the Art of Comparing Algorithms. *Journal of Experimental Algorithmics*, 7:1, 2002.