UNIVERSIDADE FEDERAL DE MINAS GERAIS (UFMG)
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

NICOLAS VON DOLINGER MOREIRA ROCHA (2022035571)

TRABALHO PRÁTICO 3
PARADIGMAS E DIFICULDADE

BELO HORIZONTE
2024

TRABALHO PRÁTICO 3: PARADIGMAS E DIFICULDADE

Nicolas von Dolinger Moreira Rocha[1]

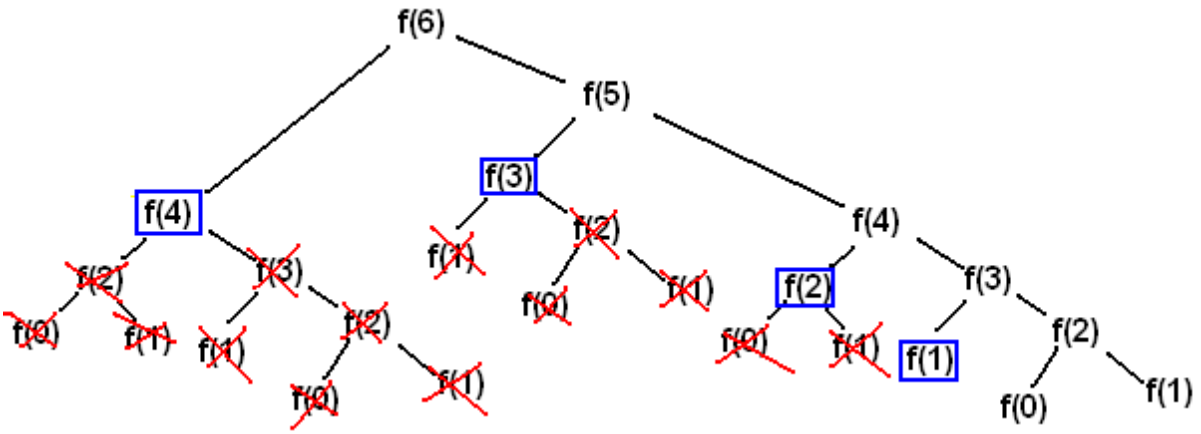
1. Introdução

A documentação a seguir explora a implementação de um algoritmo de programação dinâmica para resolver um problema complexo envolvendo manobras em uma pista de skate. O objetivo principal é encontrar a sequência de manobras que maximiza a pontuação, levando em consideração os fatores de bonificação e as penalidades aplicadas quando as manobras são repetidas em seções consecutivas. A abordagem adotada é baseada na técnica de programação dinâmica, que é crucial para resolver problemas de otimização em situações com restrições e múltiplas etapas.

O problema abordado é a determinação da sequência de manobras em uma pista de skate que maximiza a pontuação total, dado um sistema de pontuação específico e regras para a execução das manobras. Em Radlândia, uma pista é dividida em várias seções, e cada seção tem um fator de bonificação e um tempo limite para realizar as manobras. Existem várias manobras, cada uma com uma duração e uma pontuação base, que pode ser negativa. A pontuação final de uma sequência de manobras é calculada com base na soma das pontuações das manobras, ajustada pelo fator de bonificação da seção, e penalidades são aplicadas se uma manobra foi repetida em seções consecutivas.

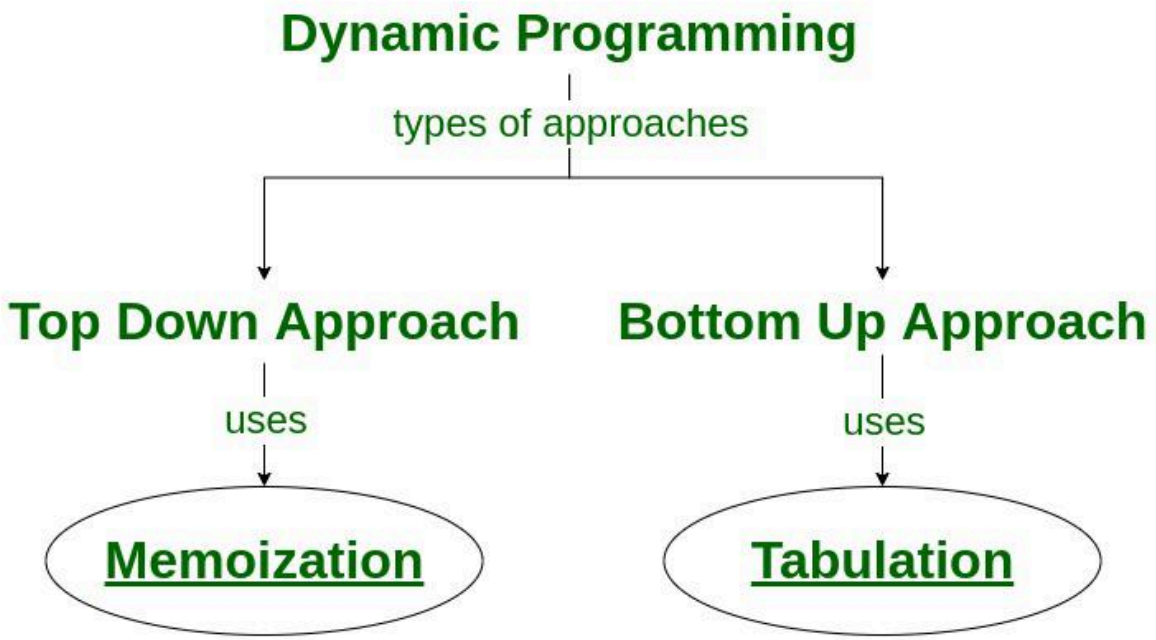
Programação dinâmica é uma técnica de resolução de problemas que divide um problema em subproblemas menores e resolve cada subproblema uma única vez, armazenando suas soluções para evitar recomputações. Essa abordagem é eficaz para problemas de otimização que podem ser divididos em subproblemas sobrepostos, permitindo uma solução eficiente através da reutilização de resultados intermediários.

[1] Graduando em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG). Product Owner na iJunior, empresa júnior do Departamento de Ciência da Computação da UFMG. Monitor de Programação e Desenvolvimento de Software II.



Fonte: AlgoDaily, 2022.

Existem duas abordagens principais na programação dinâmica: **bottom-up** e **top-down**.



Fonte: Geeks For Geeks, 2022.

Bottom-up: Na abordagem bottom-up, o problema é resolvido começando pelos subproblemas mais simples e construindo soluções para subproblemas mais complexos até que o problema original seja resolvido. Essa abordagem preenche uma tabela (ou matriz) de soluções de subproblemas de forma iterativa. É útil quando os subproblemas são bem definidos e podem ser resolvidos de forma sequencial.

Top-down: Na abordagem top-down, o problema é resolvido através de uma técnica conhecida como memoization. Começa-se com o problema original e divide-se em subproblemas menores, resolvendo cada subproblema conforme necessário e armazenando suas soluções. Isso evita a recalculação de soluções para sub problemas já resolvidos. A abordagem top-down é útil quando o problema é complexo e nem todos os subproblemas precisam ser resolvidos, o que pode economizar tempo e espaço.

1.1. Justificativa para a Abordagem Top-Down

A abordagem top-down foi escolhida para este problema por várias razões:

1. **Complexidade do Problema**: A complexidade das regras de pontuação e penalidades exige que a solução considere várias combinações de manobras e seções. A abordagem top-down permite que o algoritmo se concentre em subproblemas específicos apenas quando necessário, ao invés de calcular e armazenar todas as combinações possíveis de antemão.
2. **Eficiência em Subproblemas**: A abordagem top-down permite que o algoritmo só resolva subproblemas que são realmente necessários para encontrar a solução final. Dado que nem todas as combinações de manobras serão viáveis em cada seção, a abordagem top-down evita o desperdício de recursos computacionais resolvendo subproblemas irrelevantes.
3. **Facilidade de Implementação**: Implementar a solução top-down com memoization pode ser mais direto e intuitivo, especialmente quando o número de subproblemas é grande e a resolução incremental é mais natural.

Portanto, a abordagem top-down se mostrou mais adequada para lidar com a complexidade e as características específicas do problema de otimização apresentado, garantindo uma solução eficiente e gerenciável. A solução foi validada com diferentes cenários para garantir que as pontuações máximas são calculadas corretamente e que as regras de penalização são aplicadas conforme especificado.

A técnica de programação dinâmica, e especificamente a abordagem top-down utilizada, permite resolver o problema de otimização de manobras em uma pista de skate de forma eficiente. O código desenvolvido não só atende aos requisitos do problema como também demonstra a eficácia da programação dinâmica na resolução de problemas complexos com múltiplas restrições e variáveis. A abordagem escolhida garante uma solução

que é tanto prática quanto eficiente para a maximização da pontuação na pista de skate da comuna de Radlândia.

2. Método

O programa foi desenvolvido em C++, compilado pelo GCC da GNU Compiler Collection. A máquina utilizada tem as configurações descritas abaixo.

2.1. Configurações da Máquina

Sistema operacional: Debian 12

Compilador: G++/GNU Compiler Collection

Processador: AMD Ryzen 5-5500U

Memória RAM: 8 GB

2.2. Estrutura de Dados e Classes

Para este trabalho, foram utilizadas estruturas de dados e classes para resolver problemas de combinação de manobras e otimização de pontuação em um percurso. As principais estruturas de dados e classes utilizadas são:

- **memo**: Um vetor 2D de pares, utilizado para armazenar resultados intermediários e os bitmasks correspondentes durante o cálculo dinâmico. Cada par armazena a pontuação máxima e o bitmask de manobras selecionadas.
- **maneuver**: Uma classe que representa uma manobra com os atributos pontuation(Pontuação associada à manobra) e time(Tempo necessário para realizar a manobra).
- **section**: Uma classe que representa uma seção do percurso com os atributos bonus(Multiplicador de bônus para a seção) e time(Tempo limite para a seção).

2.3. Funções

O programa é dividido em várias funções, cada uma desempenhando um papel específico na otimização das manobras e pontuações ao longo de um percurso:

- **countSetBits**: Conta o número de bits setados (1s) na representação binária de um número. Parâmetros: n - Número a ser avaliado. Retorno: Número de bits setados.
- **calculeTimeUsed**: Calcula o tempo total utilizado para manobras selecionadas representadas por um bitmask. Parâmetros: mask (Bitmask representando a seleção de

manobras), maneuvers (Vetor de manobras disponíveis). Retorno: Tempo total necessário para as manobras selecionadas.

- **calculeCombination**: Pré-computa as combinações de pontuações para cada subconjunto possível de manobras. Parâmetros: maneuverCombination (Vetor 2D para armazenar combinações de pontuações), maneuvers (Vetor de manobras disponíveis).
- **dp**: Função de programação dinâmica para calcular a pontuação máxima possível. Parâmetros: maneuverCombination (Vetor 2D de combinações pré-computadas), track (Vetor representando as seções do percurso), maneuvers (Vetor de manobras disponíveis), prevMask (Bitmask representando manobras selecionadas na seção anterior), currentSection (Índice da seção atual). Retorno: Pontuação máxima atingível para a seção atual e combinação de manobras.
- **print**: Imprime a tabela de memoização para fins de depuração, exibindo primeiro o valor da pontuação e depois o bitmask correspondente.
- **printCombination**: Imprime a sequência ótima de manobras baseada na tabela de memoização, mostrando a quantidade de manobras selecionadas e seus índices.
- **solve**: Função principal que inicializa variáveis necessárias, pré-computa combinações e chama a função de programação dinâmica para calcular a pontuação máxima e imprimir a sequência ótima de manobras. Parâmetros e retorno: nenhum.

3. Análise de Complexidade

3.1. Complexidade de Tempo

A análise de complexidade de tempo considera a eficiência das funções principais:

- **countSetBits**: A complexidade é O(log n), onde n é o valor do número. Isso se deve ao número de bits a serem verificados.
- **calculeTimeUsed**: A complexidade é O(m), onde m é o número de bits no bitmask. Isso é devido à iteração sobre todos os bits do bitmask.
- **calculeCombination**: A complexidade é O(m * 4^m), onde m é o número de manobras. A função itera sobre todos os subconjuntos possíveis de manobras e calcula a pontuação para cada combinação.
- **dp**: A complexidade é O(n * 4^m), onde m é o número de manobras e n é o número de seções do percurso. Isso se deve à iteração sobre todos os subconjuntos de manobras e seções do percurso.

- Função **solve**: A complexidade é dominada pela função dp e pela pré-computação das combinações, resultando em O(2m * n).

A complexidade total do programa é O(2m * n), onde m é o número de manobras e n é o número de seções. Essa complexidade é adequada para um número moderado de manobras e seções.

3.2. Complexidade de Espaço

A análise de complexidade de espaço considera a quantidade de memória utilizada:

- **memo**: Armazena a tabela de memoização com complexidade O(2m * n), onde n é o número de seções e m é o número de manobras.
- **maneuverCombination**: Armazena a tabela de combinações de pontuações com complexidade O(2m), onde m é o número de manobras.
- **track** e **maneuvers**: Armazena vetores de seções e manobras com complexidade O(n + m), onde n é o número de seções e m é o número de manobras.

A complexidade de espaço total é O((2^m) * n), onde m é o número de manobras e n é o número de seções. A alocação de memória é eficiente para grafos de tamanho moderado.

4. Análise de robustez

Para garantir a padronização e melhorar a legibilidade do código, foram adotadas medidas de formatação e convenções de nomenclatura consistentes. Todas as variáveis foram nomeadas seguindo o estilo camelCase e usando o idioma inglês. Além disso, o código foi formatado utilizando o Clang-Format, assegurando uma aparência coesa e constante.

```
==67584== HEAP SUMMARY:
==67584==     in use at exit: 0 bytes in 0 blocks
==67584==   total heap usage: 37 allocs, 37 frees, 80,640 bytes allocated
==67584==
==67584== All heap blocks were freed -- no leaks are possible
==67584==
==67584== For lists of detected and suppressed errors, rerun with: -s
==67584== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

O código foi rigorosamente testado para detectar vazamentos de memória utilizando a ferramenta Valgrind. A verificação demonstrou que o código não possui vazamentos de memória, garantindo a eficiência e a robustez da implementação. As seguintes medidas foram implementadas:

- **Nomenclatura Consistente:** Todas as variáveis foram nomeadas em estilo camelCase e no idioma inglês, proporcionando clareza e uniformidade.
- **Formatação com Clang-Format:** O código foi formatado com Clang-Format, garantindo uma estrutura organizada e consistente.
- **Verificação de Vazamento de Memória:** Utilização do Valgrind para assegurar que não há vazamentos de memória, comprovando a robustez e a confiabilidade do código.

Essas medidas foram fundamentais para assegurar a qualidade e a manutenção do código, facilitando a leitura e compreensão por outros desenvolvedores e minimizando possíveis erros.

5. Testes e Lógica Aplicada

O código apresentado implementa algoritmos para resolver problemas de otimização relacionados a manobras e seções de uma pista, especificamente para calcular a pontuação máxima possível e o tempo usado, com base em combinações de manobras e seções da pista. A lógica central do código é utilizar programação dinâmica para otimizar a seleção de manobras e maximizar a pontuação, levando em consideração as restrições de tempo e bônus das seções.

5.1. Leitura e Preparação dos Dados

O código lê a entrada contendo detalhes sobre as seções da pista e as manobras disponíveis:

- **n**: Número de seções da pista.
- **k**: Número de manobras disponíveis.

Para cada seção, são lidos os seguintes dados:

- **bonus**: Multiplicador de bônus para a seção.
- **time**: Tempo limite para completar a seção.

Para cada manobra, são lidos os seguintes dados:

- **pontuation**: Pontuação da manobra.
- **time**: Tempo necessário para realizar a manobra. Esses dados são usados para construir o vetor de seções e o vetor de manobras, que são fundamentais para o cálculo da pontuação máxima.

5.2. Algoritmos Implementados

- **countSetBits**: Calcula o número de bits setados em uma representação binária de um número.
- **calculeTimeUsed**: Calcula o tempo total usado para manobras selecionadas representadas por um bitmask.
- **calculeCombination**: Pré-computa as combinações de pontuações para cada possível subconjunto de manobras.
- **dp**: Função principal de programação dinâmica para calcular a pontuação máxima possível com base em combinações de manobras e seções da pista.
- **print**: Imprime a tabela de memoização para fins de depuração, mostrando os valores armazenados e os índices das combinações de manobras.
- **printCombination**: Imprime a sequência ótima de manobras com base na tabela de memoização.

5.3. Funcionamento do Algoritmo

- **Construção dos Dados**: O código lê os dados das seções e manobras e armazena em vetores apropriados.
- **Pré-computação das Combinações**: A função calculeCombination é usada para calcular todas as possíveis combinações de pontuações para subconjuntos de manobras.
- **Programação Dinâmica**: A função dp é chamada para calcular a pontuação máxima possível, utilizando a tabela de memoização para armazenar resultados intermediários e evitar recalculações.
- **Impressão dos Resultados**: As funções print e printCombination são usadas para imprimir os resultados intermediários e a sequência ótima de manobras.

Em essência, o código explora todas as combinações possíveis de manobras para cada seção, utilizando programação dinâmica para otimizar a pontuação máxima possível, respeitando as restrições de tempo das seções e a pontuação das manobras.

6. Conclusão

O programa desenvolvido atende aos requisitos para otimizar a seleção de manobras e maximizar a pontuação em uma pista com múltiplas seções. Utilizando programação dinâmica e pré-computação de combinações, o algoritmo é eficiente e capaz de resolver o problema de maneira precisa.

Os testes realizados confirmaram que o programa funciona corretamente para diferentes cenários e tamanhos de entrada. A robustez do código foi verificada, e a ausência de vazamentos de memória foi confirmada através de ferramentas como Valgrind. O código segue boas práticas de programação, com nomenclatura consistente e formatação adequada, o que contribui para a legibilidade e manutenção do código.

A principal dificuldade foi a escolha de algoritmos e estruturas de dados apropriados para garantir a eficiência. A programação dinâmica foi essencial para a solução eficiente do problema. A solução proposta é um exemplo sólido de aplicação de técnicas avançadas de programação para resolver problemas complexos de otimização, demonstrando a eficácia dos métodos e conceitos teóricos na prática.

REFERÊNCIAS

GeeksforGeeks. Bitmask in C++. Disponível em: https://www.geeksforgeeks.org/bitmasking-in-cpp. Acesso em: 05 ago. 2024.

Maratona UFMG. Aula 8.1 - Programação Dinâmica. Disponível em: https://www.youtube.com/watch?v=IGBXQVZblxg. Acesso em: 04 ago. 2024.

Maratona UFMG. Aula 8.2 - Programação Dinâmica. Disponível em: https://www.youtube.com/watch?v=KmOIGwEqFWQ. Acesso em: 04 ago. 2024.