Alocação otimizada de aulas em diferentes salas utilizando o Método Guloso

Alanderson Sousa Lopes
Gabriel Alexandre Alovisi
Pablo Gabriel Sustisso
Pedro Henrique Câmara
Samuel Grontoski

***Resumo*—O objetivo deste estudo é abordar o problema da alocação otimizada de aulas em diferentes salas utilizando o Método Guloso e a linguagem de programação Python. Este problema é altamente relevante em ambientes acadêmicos, onde a gestão eficiente do espaço pode beneficiar significativamente tanto os alunos quanto os professores. A solução proposta envolve o desenvolvimento de um sistema que distribui as aulas disponíveis nas salas de forma eficiente, considerando restrições específicas como o tamanho da turma, horários de início e término, e a capacidade das salas.**

***Palavras Chave*—Algoritmo, Alocação, Código, Complexidade, Método, Otimização, Python.**

I. Introdução

O presente relatório, elaborado para o trabalho final da disciplina de Algoritmos e Estruturas de Dados 2, do curso de Engenharia de Computação da UTFPR Campus Pato Branco, tem como objetivo explicitar a resolução do problema escolhido pelos docentes: "Alocação otimizada de aulas em diferentes salas". O objetivo consiste na criação de um sistema que distribua de forma eficiente as disciplinas disponíveis nas salas de aula, considerando que cada disciplina possui uma descrição específica, o número de alunos e horários de início e término. As salas de aula, por sua vez, possuem identificações e capacidades máximas de alunos que podem acomodar. A solução deve garantir que todas as disciplinas sejam alocadas em salas adequadas, respeitando as capacidades e horários, e deve também minimizar a quantidade de salas de aula utilizadas,

A escolha deste problema foi pela sua relevância e aplicabilidade prática, uma vez que reflete uma situação real vivenciada no cotidiano universitário. A implementação de uma solução para este problema poderia ser diretamente aplicada na gestão de espaços da universidade, melhorando a organização e utilização das salas de aula, e beneficiando todos os participantes do grupo e a comunidade acadêmica como um todo.

A linguagem escolhida para a elaboração do programa foi Python, por ser uma linguagem eficiente e de fácil aprendizagem. Além disso, os integrantes do grupo têm interesse em aprofundar seus conhecimentos na mesma, o que tornou a escolha ainda mais relevante.

II. Descrição do problema e motivações para a escolha das estratégias

O problema central abordado neste trabalho é a alocação otimizada de aulas em diferentes salas de aula. Este problema é de grande relevância prática e consiste em distribuir as disciplinas disponíveis entre as salas de aula de forma eficiente, considerando diversas restrições e objetivos. Para isso, são fornecidas duas listas principais: uma lista de disciplinas e uma lista de salas de aula.

Cada disciplina possui as seguintes características:

- **Descrição:** Um texto que descreve o conteúdo ou título da disciplina.
- **Tamanho da turma:** O número de alunos que estão matriculados na disciplina.
- **Horário de início:** O horário em que a disciplina começa.
- **Horário de término:** O horário em que a disciplina termina.

Cada sala de aula possui as seguintes características:

- **Identificação:** Um identificador único que diferencia cada sala de aula.
- **Capacidade:** O número máximo de alunos que a sala pode acomodar.

O desafio consiste em alocar as disciplinas nas salas de aula de maneira que satisfaça as seguintes condições:

1) **Capacidade:** Nenhuma sala deve exceder sua capacidade máxima de alunos.
2) **Horários:** As disciplinas alocadas na mesma sala não devem ter sobreposição de horários.
3) **Otimização de Recursos:** A quantidade total de salas utilizadas deve ser minimizada, ou seja, o objetivo é utilizar o menor número possível de salas sem deixar de atender às necessidades das disciplinas.

Para resolver este problema, é necessário desenvolver um algoritmo que considere todas essas restrições e encontre uma solução que maximize a eficiência na utilização das salas de aula. Este algoritmo deve ser capaz de lidar com diferentes combinações de horários e capacidades, garantindo sempre a melhor alocação possível.

A. Principais motivações

A escolha das estratégias para resolver o problema de alocação otimizada de aulas em salas de aula foi guiada

por uma série de motivações teóricas e práticas. Primeiramente, a relevância do problema em contextos reais de gestão acadêmica motivou a busca por uma solução eficiente e aplicável. A organização adequada dos horários e espaços de aulas é um desafio comum em instituições de ensino, e uma solução eficaz pode trazer benefícios significativos para estudantes, professores e administradores.

B. Simplicidade e Eficácia do Método Guloso

Optamos por utilizar o método guloso (greedy algorithm) como a principal estratégia de alocação devido à sua simplicidade e eficácia. Este método é conhecido por ser eficiente em termos de tempo de execução e por produzir soluções satisfatórias para uma ampla gama de problemas de otimização combinatória. A principal vantagem do método guloso é que ele toma decisões locais ótimas com a esperança de que essas decisões levarão a uma solução globalmente ótima. No contexto do nosso problema, isso significa que o algoritmo sempre tentará alocar uma disciplina na melhor sala disponível naquele momento, de acordo com critérios pré-definidos.

C. Implementação em Python

A linguagem de programação Python foi escolhida para a implementação do algoritmo devido à sua sintaxe clara e expressiva, que facilita o desenvolvimento e a manutenção do código. Python também oferece uma vasta gama de bibliotecas e ferramentas que podem ser utilizadas para ler dados, manipular horários e capacidades, e criar interfaces gráficas. Especificamente, utilizamos a biblioteca Tkinter para desenvolver a interface gráfica do usuário, permitindo uma interação amigável e intuitiva com o sistema.

Além disso, a escolha por Python foi influenciada pelo fato de que a maioria dos membros do grupo já possui experiência com a linguagem e está interessada em aprofundar seus conhecimentos. Isso não só torna o processo de desenvolvimento mais eficiente, como também contribui para o aprendizado e crescimento pessoal dos integrantes do grupo.

D. Manipulação de Dados e Facilidades de Entrada e Saída

Para facilitar a entrada de dados no sistema, implementamos funções que leem as disciplinas e salas de arquivos de texto (.txt). Essas funções, denominadas *'ler_disciplinas'* e *'ler_salas'*, processam as informações dos arquivos e criam listas das classes Disciplina e Sala, respectivamente. Essa abordagem foi escolhida por sua simplicidade e por facilitar a integração com outras partes do sistema. Além disso, a função *'hora_para_minutos'* foi desenvolvida para converter horários no formato HH:MM para minutos, o que simplifica a comparação e manipulação dos horários durante o processo de alocação.

E. Otimização da Alocação

A função principal, *'alocar_disciplinas'*, é responsável por alocar as disciplinas nas salas de aula. Utilizando o método guloso, o algoritmo ordena inicialmente as listas de disciplinas e salas de aula. As disciplinas são ordenadas pelo horário de término, enquanto as salas são ordenadas pela sua capacidade. Esta ordenação é crucial para garantir que as disciplinas sejam alocadas de forma eficiente, respeitando as capacidades das salas e evitando sobreposição de horários. Durante a execução do algoritmo, para cada disciplina, o sistema verifica se há uma sala disponível que possa acomodar a turma e cujo horário esteja livre. Se encontrar uma sala adequada, a disciplina é alocada nessa sala e o horário de término da última disciplina alocada é atualizado. Caso contrário, a disciplina é adicionada a uma lista de disciplinas não alocadas.

Essas escolhas estratégicas foram motivadas pela necessidade de encontrar uma solução prática e eficiente para o problema de alocação de aulas, refletindo a importância de uma boa gestão dos recursos acadêmicos e o impacto positivo que isso pode ter na qualidade do ensino e na organização das instituições educacionais.

As estratégias e motivações descritas anteriormente oferecem uma visão abrangente da abordagem adotada para solucionar o problema de alocação otimizada de aulas. Utilizando o método guloso e a implementação em Python, buscamos uma solução eficiente e prática que atende às necessidades das instituições de ensino. No próximo tópico, detalharemos cada uma das funções mencionadas, explicando seu funcionamento e como contribuem para a resolução do problema. As funções *'ler_disciplinas'*, *'ler_salas'*, *'hora_para_minutos'* e 'alocar_disciplinas' serão exploradas em profundidade, demonstrando sua importância e aplicação no contexto do nosso algoritmo de alocação.

III. Descrição das soluções do problema

Visando resolver o problema de alocação de disciplinas em salas de aula de forma eficiente, utilizamos o método guloso no intuito de buscar a solução ideal. Este código foi implementado utilizando Python e a biblioteca Tkinter para a interface gráfica. Abaixo, apresentamos uma descrição detalhada das soluções implementadas para abordar esse problema.

A. Definição das Classes

A primeira parte do nosso código consiste na definição de duas classes principais: Disciplina e Sala. A classe Disciplina representa cada disciplina com suas características principais, como descrição, tamanho da turma, hora de início e hora de fim. A classe Sala representa cada sala de aula, contendo informações sobre a capacidade e as disciplinas alocadas.

B. Leitura e Conversão de Dados

Para facilitar a entrada de dados, foram implementadas funções que leem as disciplinas e as salas de arquivos .txt. As funções 'ler_disciplinas' e 'ler_salas' leem as informações dos arquivos e criam listas das classes Disciplina e Sala, respectivamente. Além disso, a função 'hora_para_minutos' converte o horário no formato HH:MM para minutos, facilitando a comparação de horários durante o processo de alocação, já que todos os valores passarão a ser inteiros.

C. Algoritmo de Alocação

A função 'alocar_disciplinas' é responsável por alocar as disciplinas nas salas de aula. Como utilizamos o método guloso para a implementação deste algoritmo, é importa termos o cuidado em como os dados chegarão ao mesmo. Por isso, é realizada a ordenação de forma crescente das duas listas, disciplinas e salas, antes da chamada do algoritmo. A lista de disciplinas é ordenada pelo horário de término das disciplinas, enquanto a lista de salas é ordenada pela capacidade das salas. Na execução do algoritmo, é considerada a capacidade das salas e os horários das disciplinas para garantir que não haja sobreposição de horários e que a capacidade da sala não seja excedida. O algoritmo segue os seguintes passos:

- Para cada disciplina, verifica se há uma sala disponível que possa acomodar a turma e cujo horário esteja livre.
- Se encontrar uma sala adequada, aloca a disciplina nessa sala e atualiza o horário de término da última disciplina alocada na sala.
- Caso não encontre uma sala disponível, a disciplina é adicionada à lista de disciplinas não alocadas.

D. Interface Gráfica

Para tornar a visualização dos resultados obtidos mais amigável, foi desenvolvida uma interface gráfica utilizando Tkinter. A interface permite visualizar a alocação das disciplinas nas salas de aula, e ao final, exibe as disciplinas que não puderam ser alocadas.

E. Exibição de Dados

A função 'exibir_alocacao' é responsável por atualizar a árvore com as disciplinas alocadas e não alocadas. Esta função organiza os dados de forma clara para facilitar a visualização dos resultados.

F. Visão Geral

A solução proposta para a alocação otimizada de disciplinas em salas de aula aborda de maneira eficiente os desafios de capacidade e sobreposição de horários. A combinação de um algoritmo bem estruturado e uma interface gráfica intuitiva proporciona uma ferramenta prática e eficaz para a gestão de salas de aula.

IV. Análise de Complexidade de Tempo e de Espaço de cada solução

Analisar a complexidade de um código pode parecer uma tarefa complexa dependendo da aplicação e de como a linguagem de programação usada foi primordialmente estruturada, que neste caso foi utilizado a linguagem Python.

Para fazermos uma análise precisa neste caso, compreender a complexidade dos métodos incorporados (built-in) em Python é crucial para fazer uma análise completa da complexidade do código. Dito isso, vamos detalhar cada função e determinar a complexidade individualmente. Em seguida, veremos como elas se combinam quando o programa é executado.

A. Funções e Complexidade de Tempo

- **hora_para_minutos(hora):** Função que converte uma string de hora no formato HH:MM para minutos. A operação principal aqui é o método *'strip()'* da string e a conversão para inteiros (cujo é uma operação constante). O método *'strip()'* remove os espaços em branco do início e do final de uma string. A complexidade desse método depende do comprimento da string, pois ele precisa verificar cada caractere desde o início e o final até encontrar um caractere que não seja um espaço em branco. Portanto, a complexidade é ***O(n)***, onde n é o comprimento da string.

- **ler_disciplinas(arquivo):** Essa função lê as disciplinas de um arquivo e cria instâncias de *'Disciplina'*. Lê cada linha do arquivo: ***O(n)*** onde ***n*** é o número de linhas (disciplinas). Para cada linha, faz *'split()'* e cria uma instância de *'Disciplina'*: ***O(m)***, onde ***m*** é o tamanho da string. Complexidade total: ***O(n * m)***.

- **ler_salas(arquivo):** Essa função lê as salas de um arquivo e cria instâncias de *'Sala'*. Lê cada linha do arquivo: ***O(n)*** onde ***n*** é o número de linhas (disciplinas). Para cada linha, faz *'split()'* e cria uma instância de *'Sala'*: ***O(m)***, onde ***m*** é o tamanho da string. Complexidade total: ***O(n * m)***.

- **alocar_disciplinas(disciplinas, salas):** Essa função aloca as disciplinas em salas.
  - **Ordenação das Disciplinas:** *'disciplinas.sort(key=lambda x: x.hora_fim)'*: O método *'sort()'* do Python utiliza um algoritmo chamado Timsort que é um algoritmo de ordenação híbrido derivado do merge sort e do insertion sort, projetado para ter boa performance em vários tipos de dados do mundo real[3]. O timsort possui complexidade ***O(n log n)***[3].
  - **Ordenação das Salas:** *'salas.sort(key=lambda x: x.capacidade)'*: ***O(m log m)***.
  - **Alocação das Disciplinas:** Laço externo sobre as disciplinas: ***O(n)***. Para cada disciplina, há um laço interno sobre as salas: ***O(m)***. No pior caso, cada disciplina percorre todas as salas para encontrar uma alocação. Complexidade total: ***O(n * m)***.

- **Ordenação Final das Salas pelo identificador:** *'salas.sort(key=lambda x: x.identificacao)'*: ***O(n log n)***.

- **Impressão dos dados finais:** ***O(n * m)***.

- **Complexidade Total da Função:** $O(n) + O(n * m) + O(n * m) + O(n \log n) + O(m \log m) + O(n * m) + O(n * m)$. Simplificando, considerando que $n * m$ tende

a dominar em situações comuns, a complexidade total será ***O(n * m)***.

B. Funções e Complexidade de Espaço

Classe *'Disciplina'* e *'Sala'*: Cada instância dessas classes ocupa espaço proporcional aos atributos que elas armazenam. Considerando uma instância típica:

- ***Disciplina:***
  - **descricao:** Uma string de comprimento ***d***.
  - **tamanho_turma:** Um inteiro.
  - **hora_inicio:** Uma string de comprimento 5 (HH:MM)
  - **hora_fim:** Uma string de comprimento 5 (HH:MM)

- ***'Sala':***
  - **identificacao:** Uma string de comprimento ***i***.
  - **capacidade:** Um inteiro.
  - **disciplinas:** Uma lista de instâncias *'Disciplina'*.

- **hora_para_minutos(hora):** Esta função apenas manipula strings e inteiros temporários. O espaço é constante ***O(1)***.

- **ler_disciplinas(arquivo):** Esta função lê as disciplinas de um arquivo e cria uma lista de instâncias de *'Disciplina'*. Lista disciplinas com n instâncias de *'Disciplina'*. Para cada linha lida, armazenamos temporariamente uma string e fazemos um *'split'*, criando uma lista de substrings temporárias. Complexidade de espaço: ***O(n * d)*** se considerarmos d como o comprimento médio da descrição e outros atributos fixos.

- **ler_salas(arquivo):** Esta função lê as salas de um arquivo e cria uma lista de instâncias de *'Sala'*. Lista *'salas'* com ***m*** instâncias de *'Sala'*. Para cada linha lida, armazenamos temporariamente uma string e fazemos um *'split'*, criando uma lista de substrings temporárias. Complexidade de espaço: ***O(m * i)*** se considerarmos ***i*** como o comprimento médio da identificação e outros atributos fixos.

- **Ordenações:** Não requer espaço adicional significativo além de uma pequena quantidade para a ordenação (Timsort).

- **Alocação:** Lista fim_salas de tamanho ***m***: ***O(m)***. Durante a alocação, não criamos novos objetos *'Disciplina'* ou *'Sala'*, apenas modificamos a lista de disciplinas dentro de cada sala.

- **Complexidade Total da Função:** a complexidade de espaço total é dominada pelo armazenamento das disciplinas e salas: ***O(n * d + m * i)***. Isso reflete o espaço necessário para armazenar todas as disciplinas e salas, bem como a lista adicional utilizada durante a alocação.

C. Conclusão de Eficiência

Ao analisar a complexidade de um código Python, é essencial conhecer a complexidade dos métodos incorporados que são chamados. Isso permite uma análise precisa da complexidade total do código.

No caso do código fornecido, a complexidade total é ***O(n * m)***, onde ***n*** é o número de disciplinas e ***m*** é o número de salas. Isso significa que o tempo de execução cresce linearmente com o produto do número de disciplinas e salas.

Além disso, a complexidade de espaço do código é dominada pelo armazenamento das disciplinas e salas, resultando em ***O(n * d + m * i)***, onde ***d*** é o comprimento médio das descrições das disciplinas e ***i*** é o comprimento médio das identificações das salas. Isso reflete o espaço necessário para armazenar todas as disciplinas e salas, bem como a lista adicional utilizada durante a alocação.

Portanto, a análise de complexidade de tempo e espaço mostra que o programa consome recursos proporcionalmente ao número de disciplinas e salas, e ao tamanho dos atributos dessas entidades. Garantir a eficiência do código é crucial, especialmente ao lidar com grandes volumes de dados, para manter o desempenho e a utilização de memória em níveis aceitáveis em aplicações de grande escala.

V. Conclusão

No geral, o desenvolvimento de um sistema para alocação otimizada de aulas em salas de aula universitárias apresenta soluções para melhorar a organização acadêmica. Dessa forma, escolhendo a linguagem Python para implementação, o grupo considerou sua eficiência e facilidade de aprendizado comparado a outras linguagens, alinhando-se aos interesses e capacidades dos membros da equipe.

A solução proposta pelo grupo visa não apenas alocar disciplinas em salas de maneira eficiente, considerando capacidades e horários, mas também minimizar o número de salas utilizadas. Em suma, a análise de complexidade de tempo e espaço do código revela que a solução é eficiente, com complexidade de tempo ***O(n * m)*** e espaço ***O(n * d + m * i)***, assim, maximizando a eficiência de um algoritmo guloso.

Ao garantir que o sistema seja escalável e eficiente em termos de recursos computacionais, é possível suportar operações em larga escala sem comprometer o desempenho. Sendo assim, crucial para a gestão acadêmica, onde grandes volumes de dados e múltiplos critérios de alocação precisam ser considerados simultaneamente.

Por fim, a aplicação do sistema apresentado pode transformar de maneira significante a gestão de salas de aula no ensino superior, de maneira que promova uma melhor utilização de recursos e uma boa organização para todos os envolvidos na comunidade acadêmica.

Referências

[1] J. T. Oliva, "Complexidade de Algoritmos," Algoritmos e Estrutura de Dados 2 (AE43CP), Engenharia de Computação, Dainf, UTFPR, Campus Pato Branco, 2024.

[2] J. T. Oliva, “Técnicas de Desenvolvimento de Algoritmos,” Algoritmos e Estrutura de Dados 2 (AE43CP), Engenharia de Computação, Dainf, UTFPR, Campus Pato Branco, 2024.
[3] ”TimSort – Data Structures and Algorithms Tutorials,” GeeksforGeeks, 2023. Disponível em: geeksforgeeks.org/timsort/

VI. Declaração de autoria

- **Alanderson Sousa Lopes:** Desenvolvimento das soluções propostas e introdução do artigo.
- **Gabriel Alexandre Alovisi:** Desenvolvimento das soluções propostas e descrição do problema e motivações para a escolha das estratégias.
- **Pablo Gabriel Sustisso:** Desenvolvimento das soluções propostas e análise de Complexidade de tempo e de espaço de cada solução.
- **Pedro Henrique Câmara:** Desenvolvimento das soluções propostas e conclusão do artigo
- **Samuel Grontoski:** Desenvolvimento das soluções propostas, descrição das soluções do problema e desenvolvimento da interface gráfica do algoritmo.