Relatório da Lista 1
Algoritmos Clássicos de Busca

Henrique Carniel da Silva e Marcus Vinicius Gonzales dos Santos

31 de outubro de 2024

1 Introdução

Na realização do trabalho buscamos estratégias para implementar algoritimos *Best First Search* em C++ da maneira mais eficiente possível, e mostrar os desafios e soluções para eles. A seguir e nas próximas seções descreveremos a implementação do projeto, incluindo as estratégias utilizadas e os desafios enfrentados durante o nosso desenvolvimento.

2 Implementação

Nesta seção de implementação, descreveremos os detalhes do código implementado, a sequência e lógica que seguimos durante essa etapa de desenvolvimento.

Entrada do Programa: A primeira etapa foi a criação do *parsing* de argumentos na função *main*. Optamos em seguida por abstrair essa lógica em uma classe dedicada, chamada *InputHandler*. Essa classe trata os argumentos recebidos via terminal e de arquivos de entrada, além de lidar com potenciais erros de entrada, fornecendo mensagens de erro mais claras e informativas para o usuário.

Abstração do N-Puzzle: Inicialmente, projetamos uma classe genérica para representar um N-Puzzle de tamanho variável. Posteriormente, essa estrutura foi refinada para uma hierarquia mais especializada, com uma classe base chamada *Node* e duas subclasses específicas, *Node8* e *Node15*. Dessa forma, conseguimos otimizar o desempenho para cada variante de problema, ajustando a implementação conforme o espaço de busca exigido por cada *puzzle*.

Cada classe contém os atributos essenciais para representar um nó, incluindo:

- **Nodo pai**: referência para o nó pai do nó atual.
- **Estado**: configuração atual do puzzle.
- **Identificador de geração do nodo**: ajuda a controlar a ordem de criação.
- **Custo**: valor acumulado para alcançar o nó, definido pelo tipo do algoritmo de busca.
- **Profundidade**: profundidade do nó na árvore de busca.
- **Índice do *tile* vazio**: posição do espaço em branco.
- **Valor da heurística**: cálculo usado para estimar a distância ao objetivo.

Além desses atributos, as classes também incluem métodos essenciais para manipulação e expansão do estado, tais como geração de filhos, cálculo da distância *Manhattan*, desalocação de nós, verificação de estado objetivo, entre outros. Essas especializações facilitaram uma implementação mais eficiente e modular para cada tipo de puzzle.

Implementação dos Algoritmos de Busca: A implementação dos algoritmos de busca foi relativamente tranquila, embora alguns *bugs* tenham surgido ao longo do processo. Esses problemas foram resolvidos sem grandes dificuldades. No início, seguimos fielmente a estrutura apresentada nos slides do professor como base para os algoritmos. No entanto, foi necessário fazer adaptações para integrar o código às especificidades do nosso projeto, ajustando detalhes para otimizar a compatibilidade e o desempenho da implementação.

Otimizações de Tempo e Memória: Após boa parte dos pontos anteriores estarem quase completos, nós começamos a analisar que melhorias poderiam ser feitas em relação a eficiência e uso de memória do programa. O primeiro ponto ajustado foi a questão de alocação dos nodos, eles eram instânciados em memória e depois nunca mais eram desalocados no programa, para resolver isso criamos uma lista global com todos os nodos alocados, e no final da execução, ou em uma interrupção, nós desalocamos todos os nodos.

Em seguida, utilizando ferramentas de *profiling*, identificamos que os algoritmos recursivos, como *IDFS* e *IDA**, estavam consumindo mais memória do que esperado, o que contradizia as previsões teóricas. Ao analisar mais a fundo, percebemos que esses algoritmos precisavam desalocar os nodos logo após o retorno da chamada recursiva. Essa mudança permitiu uma economia considerável de memória.

Após resolvermos as questões de alocação e desalocação de nodos, partimos para reduzir o tamanho da classe de nodo, buscando minimizar o uso de memória. Por ser instanciada milhões de vezes durante uma execução, qualquer redução no tamanho da classe teria impacto significativo. Alteramos alguns tipos de dados e definimos atributos como estáticos e constantes, reduzindo o tamanho da classe de 108 para 48 bytes. Inicialmente, estranhamos que os atributos ocupassem 18 bytes extras devido ao *padding* de memória e à *v-table* (devido aos métodos virtuais). Reordenando os atributos, conseguimos otimizar o espaço ocupado para 40 bytes, economizando mais 8 bytes.

Para otimizar o tempo de execução, implementamos outras melhorias. Cada vez que um nodo era instanciado, o programa buscava o índice do valor zero no *array*. Alteramos essa abordagem para que o nodo herde o *blankIndex* do nodo pai, caso exista; caso contrário, calcula-se o índice da forma usual. Uma estratégia similar foi aplicada ao cálculo da heurística no 15-Puzzle. Em vez de recalcular a distância *Manhattan* para cada posição, consideramos que a heurística só varia em uma unidade a cada movimento, permitindo apenas identificar se a peça movida se aproximou ou se afastou da posição final, economizando processamento.

Outro ponto de otimização foi a representação do estado. Na versão inicial da classe genérica de N-Puzzle, o estado era representado como um *vector*, dada a variabilidade do tamanho do puzzle. Com a introdução de classes especializadas, passamos a usar *arrays* de tamanho fixo, o que melhorou a eficiência. Posteriormente, optamos por representar o estado como um único *uint64_t*, o que acelerou o *hashing* dos estados e trouxe ganhos significativos de desempenho.

3 Configuração da Máquina

Nesta seção, apresentamos as especificações da máquina utilizada para a realização dos testes dos algoritmos de busca.

3.1 Hardware

- **Processador:** AMD Ryzen 5 8600G
- **Memória RAM:** 16 GB DDR4
- **Armazenamento:** SSD de 1 TB
- **Placa Gráfica:** NVIDIA GeForce RTX 4060 Ti
- **Sistema Operacional:** Windows 10

3.2 Software

- **Versão do cpp:** 11.4.0
- **Versão do make:** GNU Make 4.3

- **Ambiente de Desenvolvimento:** WSL com Ubuntu 22.04.4 LTS

4 Resultados

Nesta seção serão apresentados os resultados da execução das instâncias do 8-Puzzle e 15-Puzzle. O processo de execução, registro da memória utilizada, e salvamento da saída em arquivos .csv de todas as instâncias foi automatizado por um *script bash*. Logo abaixo há duas tabelas que reunem as informações coletadas, a primeira traz estatisticas médias sobre a saída de todas as execuções, e a segunda apresenta o quanto cada algoritmo consumiu ao máximo de memória durante a execução. Lembrando que o máximo de memória consumida durante a execução de todas as instâncias será a instância que mais consumiu memória, pois no nosso programa, toda vez que ele acaba, mesmo por uma interrupção, desalocamos todos os nodos criados durante o processo.

Table 1: Média dos Resultados para os Conjuntos de Estados Iniciais no 8-Puzzle

Algoritmo	**Nodos Expandidos**	**Comprimento**	**Tempo (ms)**	**Heurística**	**Valor Inicial**
bfs	81459.540	22.16	22.3199	0.0	13.88
idfs	2578290.6	22.16	137.501	0.0	13.88
astar	895.00000	22.16	0.26796	10.027752	13.88
idastar	2373.0300	22.16	0.21491	10.432846	13.88
gbfs	392.42000	140.52	0.11688	6.8930464	13.88

Table 2: Máximo de memória utilizada por cada algoritmo

Algoritmo	**Máximo de memória (kbytes)**
bfs	25608
idfs	3976
astar	4504
idastar	3972
gbfs	4052

Abaixo segue uma outra tabela, no mesmo formato da primeira, a qual relata agora os valores referentes a execução do conjunto de estados iniciais do 15-Puzzle. Com um *timeout* de 30 segundos para cada instância, conseguimos atingir um resultado de 86 instâncias resolvidas de um total de 100, e utilizando no máximo até 2912908 kbytes, algo em torno de 2.7Gbytes.

Table 3: Média dos Resultados para os Conjuntos de Estados Iniciais resolvidos (86/100) no 15-Puzzle

Algoritmo	**Nodos Expandidos**	**Comprimento**	**Tempo (ms)**	**Heurística**	**Heurística Inicial**
astar	4553896.92	51.83	4960.90	25.509857	36.5

5 Conclusões

Este trabalho foi uma excelente oportunidade para aplicar os algoritmos estudados em aula e explorar em profundidade as características de cada um. Além disso, nos permitiu expandir nosso conhecimento em C++, revendo aspectos importantes da linguagem e desenvolvendo uma compreensão mais consciente do uso eficiente das estruturas de dados fornecidas pela *Standard Template Library (STL)*.

A implementação também exigiu atenção a detalhes específicos, como levar em conta o gerenciamento de *virtual tables* e o alinhamento (*padding*) dos argumentos, aspectos pouco explorados em projetos práticos por nós. A otimização do desempenho em termos de tempo de execução e uso de memória se mostrou especialmente desafiadora, principalmente no caso do 15-puzzle, onde o processamento de cada instância precisava ser conduzido com máxima eficiência e controle de recursos.

Em resumo, o trabalho proporcionou uma valiosa experiência prática que reforçou e ampliou os conceitos estudados na sala de aula, contribuindo para a nossa formação técnica e preparação para desafios futuros.