Relatório da Atividade 03 - Busca Informada e Não Informada

Celso Vinícius S. Fernandes[1]

[1]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Divinópolis – MG – Brasil

[2]Bacharelado em Engenharia de Computação

celso.23@aluno.cefetmg.br

Resumo. *Este trabalho apresenta o desenvolvimento e comparação de diferentes algoritmos de busca aplicados ao problema clássico do labirinto, utilizando C++. A aplicação permite aos usuários explorar o comportamento de dois algoritmos de busca não informada (Busca em Largura - BFS e Busca em Profundidade - DFS) e dois algoritmos de busca informada (A* e Busca Gulosa) em um ambiente matricial de 10x10. A implementação mede o desempenho dos algoritmos em termos de tempo de execução, consumo de memória, completude e optimalidade, fornecendo uma análise detalhada do comportamento de cada método.*

1. Introdução

Este trabalho desenvolve e compara a eficiência de diferentes algoritmos de busca aplicados ao problema clássico do labirinto, utilizando a linguagem de programação C++. A implementação e análise dos algoritmos de busca são realizadas em um ambiente matricial de 10x10. Considerações importantes para a execução deste trabalho incluem:

1. A implementação abrange dois algoritmos de busca não informada (Busca em Largura - `BFS` e Busca em Profundidade - `DFS`) e dois algoritmos de busca informada (`A*` e Busca Gulosa - `GBFS`).
2. Cada algoritmo é avaliado em um cenário fixo de labirinto 10x10.
3. O ponto de partida e o ponto de chegada são pré-definidos, mas a disposição dos obstáculos (paredes) no labirinto pode variar.
4. Cada algoritmo deve ser capaz de encontrar o caminho do ponto de partida ao ponto de chegada, explorando todas as direções possíveis: direita, esquerda, cima e baixo.
5. O desempenho dos algoritmos é medido em termos de tempo de execução, consumo de memória, completude e optimalidade.
6. Para garantir a robustez dos resultados, cada algoritmo é executado 21 vezes no mesmo cenário, e são registrados o melhor resultado, o pior resultado, a média e o desvio padrão das medições.

A experimentação visa comparar a eficiência relativa dos diferentes algoritmos de busca em termos de recursos computacionais e precisão.

2. Metodologia

Este projeto foca na implementação e comparação de diferentes algoritmos de busca aplicados ao problema do labirinto, utilizando a linguagem de programação

C++[Deitel and Deitel 2016]. Os algoritmos são avaliados em termos de tempo de execução, consumo de memória, completude e optimalidade, proporcionando uma análise abrangente de suas eficiências. A seguir, é discutida a metodologia utilizada para desenvolver e testar esses algoritmos.

2.1. Seleção de Tecnologias

Para desenvolver a implementação com base nos requisitos, optou-se pela linguagem C++ devido à sua eficiência e familiaridade com o aluno. O projeto foi desenvolvido na plataforma Visual Studio Code para garantir uma codificação eficiente e organizada.

2.2. Métricas de Desempenho

Para avaliar o desempenho dos algoritmos de busca, foram utilizadas as seguintes métricas:

- **Tempo de Execução:** medido utilizando a biblioteca `<chrono>` do C++, que permite calcular o tempo decorrido entre o início e o fim da execução de cada algoritmo em milissegundos.
- **Consumo de Memória:** avaliado utilizando a ferramenta `Valgrind` com o subcomando `Massif`. Esta ferramenta monitora a quantidade de memória alocada dinamicamente ao longo da execução do programa.
- **Completude:** determinada pela capacidade do algoritmo em encontrar uma solução válida para o labirinto, ou seja, em alcançar o objetivo partindo do ponto inicial. Um valor de 100% indica que o algoritmo completou todas as execuções com sucesso.
- **Optimalidade:** medida comparando o comprimento do caminho encontrado por cada algoritmo com o comprimento do caminho ideal (ou mínimo). Na implementação atual, foi mensurada a partir da menor distância percorrida dentre os algoritmos e expressa como uma porcentagem do comprimento do caminho ideal.

2.3. Solução Implementada

A solução desenvolvida é dividida em classes principais que são descritas a seguir.

- `'Maze'`: Define o ambiente do labirinto, onde cada célula pode ser um caminho ou uma parede. Também armazena as posições de início e objetivo.
- `'SearchAlgorithms'`: Implementa os algoritmos de busca, incluindo Busca em Largura (BFS), Busca em Profundidade (DFS), A* e Busca Gulosa (GBFS).
- `'Utility'`: Fornece suporte para a medição de desempenho dos algoritmos e geração de relatórios, simplificando o fluxo principal de atividades.

Em primeira análise, ressalta-se a classe `'Maze'`, que modela o ambiente como uma matriz bidimensional em que cada célula pode ser um caminho livre ou uma parede. Métodos cruciais como `'isValid'` e `'isGoal'` são usados para verificar a validade de uma posição e se uma posição é o objetivo, respectivamente, simulando a percepção do agente no labirinto.

Dentro da classe `SearchAlgorithms`, cada método de busca é implementado para encontrar o caminho do início ao objetivo no labirinto, utilizando diferentes estruturas de dados e estratégias:

- O método `BFS` (Busca em Largura) utiliza uma fila (`std::queue`) para explorar os nós em níveis sucessivos, garantindo que o primeiro caminho encontrado é o mais curto em termos de número de passos. A fila permite que os nós sejam processados na ordem em que são descobertos, como mostra o pseudocódigo adiante.

```
BFS(maze):
    start = maze.getStart()
    goal = maze.getGoal()
    frontier = queue()
    frontier.enqueue(start)
    cameFrom[start] = start

    while not frontier.isEmpty():
        current = frontier.dequeue()

        if current == goal:
            return reconstructPath(cameFrom, current)

        for each neighbor in maze.getNeighbors(current):
            if neighbor not in cameFrom:
                frontier.enqueue(neighbor)
                cameFrom[neighbor] = current

    return []

```

Pseudocódigo 1. Busca em Largura (BFS)

- Já o método `DFS` (Busca em Profundidade), ilustrado no trecho a seguir, utiliza uma pilha (`std::stack`) para explorar o caminho em profundidade, movendo-se o mais longe possível ao longo de um ramo antes de retroceder. Esta implementação pode permitir que o DFS explore rapidamente os caminhos profundos, mas pode resultar em caminhos não ótimos e maior consumo de memória.

```
DFS(maze):
    start = maze.getStart()
    goal = maze.getGoal()
    frontier = stack()
    frontier.push(start)
    cameFrom[start] = start

    while not frontier.isEmpty():
        current = frontier.pop()

        if current == goal:
            return reconstructPath(cameFrom, current)

        for each neighbor in maze.getNeighbors(current):
            if neighbor not in cameFrom:
                frontier.push(neighbor)
                cameFrom[neighbor] = current

    return []

```

Pseudocódigo 2. Busca em Profundidade (DFS)

- O método `AStar` (A*) combina o custo do caminho percorrido (`g_cost`) e uma heurística estimativa do custo restante até o objetivo (`h_cost`) para guiar a busca de maneira eficiente. Utilizando uma fila de prioridade (`std::priority_queue`), o A* seleciona os nós a serem explorados com base no valor total `f_cost = g_cost + h_cost`, frequentemente buscando o caminho mais curto como é mostrado a frente.

```
AStar(maze):
    start = maze.getStart()
    goal = maze.getGoal()
    frontier = priorityQueue()
    frontier.enqueue(start, 0)
    cameFrom[start] = start
    g_cost[start] = 0

    while not frontier.isEmpty():
        current = frontier.dequeue()

        if current == goal:
            return reconstructPath(cameFrom, current)

        for each neighbor in maze.getNeighbors(current):
            new_cost = g_cost[current] + 1
            if neighbor not in g_cost or new_cost < g_cost[
   neighbor]:
                g_cost[neighbor] = new_cost
                priority = new_cost + heuristic(neighbor, goal)
                frontier.enqueue(neighbor, priority)
                cameFrom[neighbor] = current

    return []

```

Pseudocódigo 3. A*

- Em contrapartida, o `GBFS` (Busca Gulosa), apresentado abaixo, utiliza apenas a heurística (`h_cost`) para escolher o próximo nó a ser explorado, ignorando o custo acumulado do caminho percorrido. Este método, implementado também com uma fila de prioridade (`std::priority_queue`), prioriza os nós que parecem estar mais próximos do objetivo com base na heurística, o que pode levar a soluções rápidas, mas não necessariamente ótimas.

```
GreedyBestFirstSearch(maze):
    start = maze.getStart()
    goal = maze.getGoal()
    frontier = priorityQueue()
    frontier.enqueue(start, heuristic(start, goal))
    cameFrom[start] = start

    while not frontier.isEmpty():
        current = frontier.dequeue()

        if current == goal:
            return reconstructPath(cameFrom, current)

        for each neighbor in maze.getNeighbors(current):
```

```
            if neighbor not in cameFrom:
                priority = heuristic(neighbor, goal)
                frontier.enqueue(neighbor, priority)
                cameFrom[neighbor] = current

    return []

```

Pseudocódigo 4. Busca Gulosa

Para cada algoritmo, a função `'reconstructPath'` é usada para reconstruir o caminho do objetivo ao início utilizando um mapa que rastreia de onde cada nó foi alcançado. Esse método garante que o caminho final seja obtido de maneira eficiente e correta.

Ademais, a execução da simulação e a avaliação do desempenho dos algoritmos são gerenciadas pela classe `'Utility'`. A função `'measurePerformance'` é responsável por orquestrar o processo de simulação, chamando cada algoritmo de busca repetidamente e medindo o tempo de execução, consumo de memória, completude e optimalidade. Este registro das ações do algoritmo é vital para análises subsequentes, permitindo o rastreio do comportamento e desempenho dos algoritmos, como será discutido na seção de Resultados e Discussão.

3. Resultados e Discussão

3.1. Modelo e Reprodução

A fim de atestar a transparência e a reprodutibilidade dos resultados apresentados neste estudo, todo o código-fonte e a documentação detalhada deste estudo estão disponíveis em um repositório GitHub público. O diretório `'src/'` contém os scripts C++ desenvolvidos para a análise, permitindo que outros pesquisadores validem, reproduzam ou expandam a implementação. A URL do repositório Git é fornecida na seção de Referências[Celso 2024] deste artigo.

3.2. Experimentação e Análise

A seguir, são apresentados os resultados da comparação dos algoritmos de busca (`BFS`, `DFS`, `A*`, `GBFS`) em três cenários diferentes de labirinto: Simples, Sinuoso e Becos sem Saída. As configurações dos cenários estão ilustradas na Figura 1.

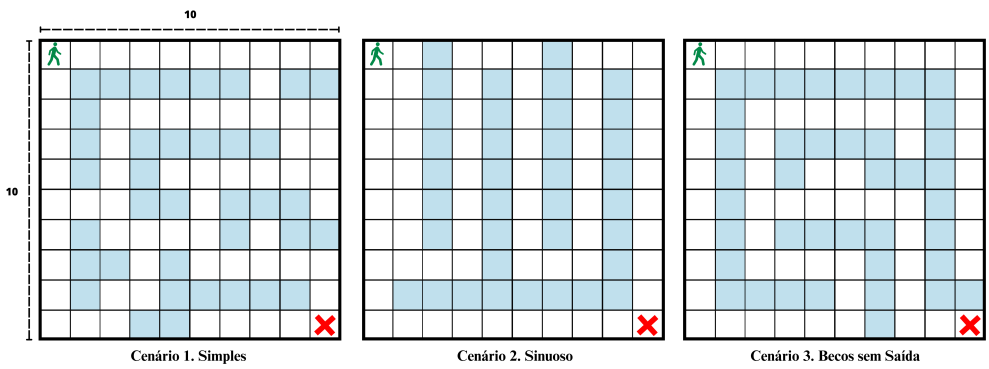

Figura 1. Ilustração das configurações dos 3 cenários utilizados nas análises.

As tabelas comparativas a frente avaliam o desempenho dos algoritmos em termos de tempo de execução, desvio padrão do tempo, melhor resultado e pior resultado para cada um dos cenários testados. Cada algoritmo foi executado 21 vezes em cada cenário, e os resultados foram calculados com base nesses testes repetidos.

Cenário 1. Simples				
Algoritmo	**Média do Tempo (ms)**	**Desvio Padrão do Tempo (ms)**	**Melhor Resultado (ms)**	**Pior Resultado (ms)**
BFS (ms)	0,55	0,04	0,54	0,69
DFS (ms)	0,56	0,02	0,55	0,66
A* (ms)	0,80	0,03	0,80	0,91
GBFS (ms)	0,37	0,01	0,36	0,39

Tabela 1. Medição do desempenho em termos de tempo de execução dos algoritmos no Cenário 1. Simples.

Cenário 2. Sinuoso				
Algoritmo	**Média do Tempo (ms)**	**Desvio Padrão do Tempo (ms)**	**Melhor Resultado (ms)**	**Pior Resultado (ms)**
BFS (ms)	0,33	0,04	0,32	0,46
DFS (ms)	0,24	0,01	0,23	0,27
A* (ms)	0,50	0,04	0,50	0,67
GBFS (ms)	0,30	0,02	0,29	0,38

Tabela 2. Medição do desempenho em termos de tempo de execução dos algoritmos no Cenário 2. Sinuoso.

Cenário 3. Becos sem Saída				
Algoritmo	**Média do Tempo (ms)**	**Desvio Padrão do Tempo (ms)**	**Melhor Resultado (ms)**	**Pior Resultado (ms)**
BFS (ms)	0,58	0,04	0,58	0,75
DFS (ms)	0,48	0,02	0,46	0,56
A* (ms)	0,83	0,03	0,83	0,92
GBFS (ms)	0,45	0,03	0,44	0,56

Tabela 3. Medição do desempenho em termos de tempo de execução dos algoritmos no Cenário 3. Becos sem Saída.

As Tabelas 1, 2 e 3 apresentam os resultados de tempo de execução dos algoritmos de busca nos três cenários. No Cenário 1 (Simples), o `GBFS` foi o mais rápido (0,37 ms), seguido por `BFS` e `DFS` com tempos semelhantes. O `A*` foi o mais lento (0,80 ms), mas todos os algoritmos completaram o cenário. No Cenário 2 (Sinuoso), o tempo de execução foi menor devido à maior eficiência dos algoritmos em um ambiente com menos bifurcações complexas. O DFS foi o mais rápido (0,24 ms), enquanto o `A*` apresentou o maior tempo novamente (0,50 ms). No Cenário 3 (Becos sem Saída), os tempos de execução aumentaram devido à complexidade adicional. Novamente, o A* foi o mais lento (0,83 ms), enquanto o `GBFS` foi o mais rápido (0,45 ms). Esses resultados sugerem que, embora o A* ofereça o caminho mais ótimo, ele faz isso com um maior custo de tempo.

Além do tempo de execução, as tabelas adiante fornecem uma análise mais abrangente do desempenho dos algoritmos, incluindo o consumo médio de memória, completude e optimalidade. Cada algoritmo foi testado nos mesmos três cenários, e as métricas foram calculadas para refletir o desempenho estimado em diferentes condições de complexidade do labirinto.

Cenário 1. Simples				
Algoritmo	**Média do Tempo (ms)**	**Cosumo Médio de Memória (KB)**	**Completude (%)**	**Optimalidade(%)**
BFS (ms)	0,55	17,09	100,00	100,00
DFS (ms)	0,56	18,12	100,00	22,00
A* (ms)	0,80	18,23	100,00	100,00
GBFS (ms)	0,37	18,59	100,00	67,00

Tabela 4. Medição do desempenho geral dos algoritmos no Cenário 1. Simples.

Cenário 2. Sinuoso				
Algoritmo	**Média do Tempo (ms)**	**Cosumo Médio de Memória (KB)**	**Completude (%)**	**Optimalidade(%)**
BFS (ms)	0,33	17,24	100,00	100,00
DFS (ms)	0,24	17,88	100,00	78,00
A* (ms)	0,50	18,32	100,00	100,00
GBFS (ms)	0,30	18,67	100,00	100,00

Tabela 5. Medição do desempenho geral dos algoritmos no Cenário 2. Sinuoso.

Cenário 3. Beco sem Saída				
Algoritmo	**Média do Tempo (ms)**	**Cosumo Médio de Memória (KB)**	**Completude (%)**	**Optimalidade(%)**
BFS (ms)	0,58	17,35	100,00	100,00
DFS (ms)	0,48	17,74	100,00	100,00
A* (ms)	0,83	18,41	100,00	100,00
GBFS (ms)	0,45	18,76	100,00	100,00

Tabela 6. Medição do desempenho geral dos algoritmos no Cenário 3. Beco sem Saída.

As Tabelas 4, 5 e 6 complementam a análise incluindo o consumo médio de memória, completude e optimalidade. No Cenário 1 (Simples), todos os algoritmos completaram o labirinto com 100% de completude, mas o A* destacou-se pela maior optimalidade, enquanto o `GBFS`, embora mais rápido, encontrou um caminho menos eficiente (67%). No Cenário 2 (Sinuoso), o `DFS` mostrou uma queda na optimalidade (78%), apesar de ser o mais rápido. O `A⋆` manteve-se como o mais preciso, mas com maior consumo de memória. No Cenário 3 (Becos sem Saída), a complexidade adicional levou a maiores tempos de execução e menor optimalidade para o `DFS` e `GBFS`, enquanto o `A⋆` continuou a ser o mais confiável, mas com maior tempo e consumo de memória.

3.3. Discussão Geral

Os resultados indicam que o `A⋆` é o algoritmo mais consistente em encontrar o caminho mais curto, embora seja o mais custoso em termos de tempo e memória. O `GBFS`, apesar de ser o mais rápido, frequentemente encontra caminhos menos eficientes. O `DFS`, embora eficiente em cenários mais simples, mostra-se menos eficaz em cenários complexos. O `BFS` se mantém como uma opção equilibrada, garantindo tanto a completude quanto a qualidade do caminho em cenários menos desafiadores.

4. Conclusão

A implementação dos algoritmos de busca para resolver o problema do labirinto demonstrou uma eficácia geral na navegação em diferentes cenários, embora com limitações evidentes dependendo da complexidade do labirinto. Observou-se que algoritmos como o `A⋆` são consistentes em encontrar o caminho mais curto, mas ao custo de maior tempo

de execução e consumo de memória, especialmente em cenários mais complexos. Por outro lado, algoritmos como o `GBFS`, embora mais rápidos, sacrificam a optimalidade, resultando em caminhos menos eficientes. A variação no desempenho dos algoritmos, conforme o cenário, reflete suas diferentes estratégias de exploração, que se adaptam melhor a certos tipos de labirinto.

Os resultados sugerem direções para aprimoramento, como a necessidade de otimizar o consumo de recursos e melhorar a qualidade do caminho encontrado em algoritmos mais rápidos como o `GBFS`. Além disso, há potencial para explorar combinações ou adaptações dos algoritmos para criar soluções mais balanceadas que possam oferecer tanto rapidez quanto precisão em uma gama mais ampla de cenários. Em resumo, enquanto cada algoritmo apresentou pontos fortes e fracos, a experimentação destaca a importância de escolher a abordagem adequada conforme as características específicas do problema de busca a ser resolvido.

5. Compilação e Execução

5.1. Escolha do cenário

No programa, o usuário pode escolher entre três cenários de labirinto predefinidos: `simpleMaze` (Cenário 1), `sinuousMaze` (Cenário 2) e `deadEndMaze` (Cenário 3), que representam diferentes níveis de complexidade. Para selecionar um desses cenários, basta alterar a linha onde a classe `Maze` é instanciada, substituindo `simpleMaze` (Padrão) pelo nome do labirinto desejado (`sinuousMaze` ou `deadEndMaze`). Essa modificação é feita diretamente no código-fonte, na função `main`. Após essa escolha, basta executar o programa que os os algoritmos de busca e sua respectiva avaliação de desempenho serão realizados no labirinto selecionado.

```
1    Maze maze(simpleMaze, start, goal);
```

Pseudocódigo 5. Linha que deve ser alterada para alternância de cenários.

5.2. Execução

Por fim, esse programa possui um arquivo `Makefile` e um `CMakeLists`, os quais realizam todo o procedimento de compilação e execução. Para tanto, cabe ao usuário escolher o de sua preferência, porém se atente ao fato de que para cada um temos as diretrizes de execução expressas nas Tabelas 7 e 8 adiante.

Makefile	
Comando	Função
make clean	Apaga a última compilação realizada contida na pasta build.
make	Executa a compilação do programa utilizando o gcc, e o resultado vai para a pasta build.
make run	Executa o programa da pasta build após a realização da compilação.
make c	Apaga, compila e executa o programa.

Tabela 7. Comandos em sequência de execução para o compilador Makefile.

CMake	
Comando	Função
cmake -B ./build	Apaga a última compilação realizada contida na pasta build.
cmake –build ./build	Executa a compilação do programa utilizando o cmake, e o resultado vai para a pasta build.
./build/app	Executa o programa da pasta build após a realização da compilação.

Tabela 8. Comandos em sequência de execução para o compilador CMake.

Referências

Celso, V. (2024). Busca informada e não informada. `https://github.com/celzin/Busca_Info_e_Nao_Info`. Acesso em: 06 ago. 2024.

Deitel, P. and Deitel, H. (2016). *Como Programar em C++*. Pearson Education, São Paulo, Brasil, 10 edition. Português.