Marcos V. Zanardi, Luige L. Figueredo
Engenharia de Computação

Escola Politécnica – PUCRS
26 de novembro de 2024

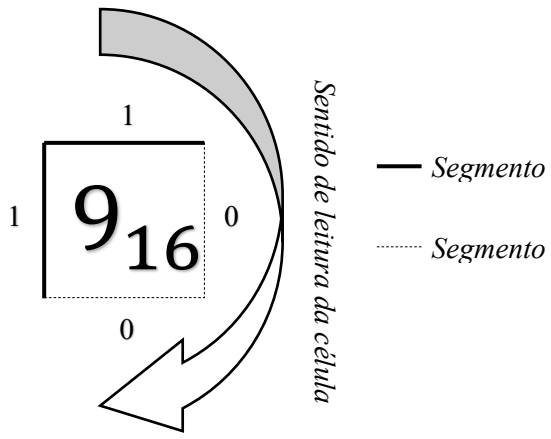
Resumo

Este artigo apresenta uma abordagem para resolver o segundo desafio da disciplina de Algoritmos e Estruturas de Dados II, do semestre 2024/2. O problema consiste em analisar um labirinto representado por células conectadas, identificando as regiões isoladas formadas por conexões válidas entre essas células. Além de quantificar essas regiões, o desafio inclui determinar qual personagem aparece com maior frequência entre elas. A solução desenvolvida utiliza técnicas eficientes de percorrimento e contagem, garantindo uma análise precisa das conexões e das ocorrências de personagens no labirinto.

I – Introdução

O desafio proposto envolve a análise de um labirinto composto por células representadas por valores hexadecimais. Quando convertidos em binário, os bits de cada valor indicam as paredes presentes na célula. Valores com caracteres alfabéticos maiúsculos representam personagens que habitam o labirinto. O objetivo é identificar as regiões isoladas, formadas por conexões válidas entre as células, e determinar qual personagem predomina em cada uma dessas regiões. A solução apresentada utiliza uma abstração do problema baseada em estruturas de grafos, onde cada célula é tratada como um nodo e as conexões entre elas são definidas pelas arestas do grafo. O ***Exemplo I*** ilustra um exemplo de célula do labirinto, apresentando os valores hexadecimais e suas configurações de paredes.

Exemplo I *: Exemplo de Célula com o valor "9"* $\rightarrow 9_{16} = 1001_2$. *Leitura iniciando na haste superior.*



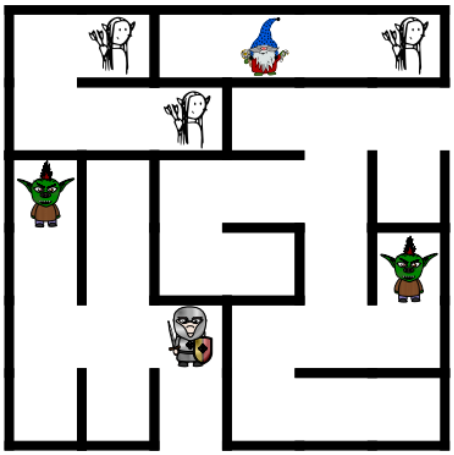
II – Abstração do Problema

A primeira característica relevante do labirinto é a sua dimensão $i \times j$, que define a quantidade de células. Cada célula contém um valor hexadecimal no intervalo $[0, F]$. A ***Figura I*** exemplifica a entrada de dados, enquanto a ***Figura II*** ilustra a representação gráfica do labirinto, destacando personagens e paredes, conforme as informações extraídas de cada elemento.

Figura I: *Labirinto com altura e largura iguais a 6, onde cada célula é representada por um valor hexadecimal, definindo suas características e conexões.*

```
6 6
9 E b A a E
3 a E b 8 8
D d 9 a 4 7
5 5 3 e 5 D
1 0 C 9 2 6
7 7 5 3 a e
```

Figura II: *Ilustração do labirinto formado a partir dos valores da matriz mostrada na Figura I, com a visualização das paredes e personagens.*



O problema pode ser descrito como a busca por regiões distintas de células conectadas e a análise de qual personagem ocorre com maior frequência em cada região.

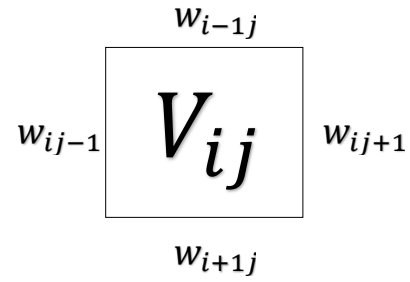
III – Estrutura do Problema

A partir das informações disponíveis sobre o labirinto, suas dimensões $i \times j$ permitem abstrair os dados de entrada para uma matriz M de células, que será utilizada como base para a construção de uma estrutura do tipo grafo.

Construção do Grafo

O grafo é estruturado de modo que cada célula da matriz M seja representada como um vértice, correspondente a um nodo. As conexões entre os vértices são definidas com base nas informações armazenadas em cada nodo, que determinam suas ligações. Conexões válidas entre células são interpretadas como arestas, estabelecendo as relações entre os vértices. Dessa forma, a matriz M é traduzida para uma estrutura de grafo que preserva integralmente as conexões e relações definidas pelos dados do problema.

O ***Exemplo II*** ilustra como as conexões de um nodo qualquer V_{ij} para outro nodo W_{ij} são respeitadas com base nos limites da matriz, mas são manipuladas dentro da estrutura do grafo



Exemplo II: *Ilustração de possíveis ligações a serem feitas em um nodo V_{ij} com outro nodo W_{ij}*

.

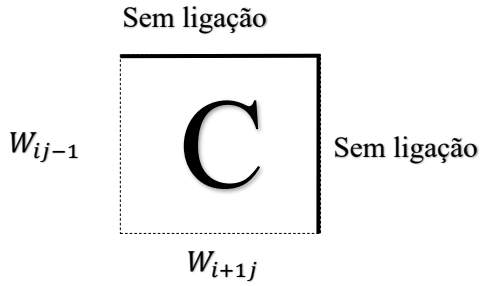
Exemplo de ligação

Para um nodo genérico V_{ij} com o valor $C_{16} = 1100_2$, as posições dos bits quando convertido para binário são:

S_3	S_2	S_1	S_0
1	1	0	0

- Sendo S_n a posição do bit – casa binária.

Quando o bit é igual a 0, significa a ausência de uma barreira naquele segmento, indicando que uma ligação entre os nodos deve ser estabelecida. Assim, o que nos interessa são apenas as posições dos bits cujo valor é 0. Para o valor de V_{ij} , cada posição binária é verificada, e, se o bit for 0, realiza-se a ligação com outro nodo W_{ij}. Nesse caso, as posições relevantes seriam S_1 e S_0. É possível observar no ***Exemplo III***, as conexões que o nodo V_{ij} deve realizar com seus adjacentes W_{ij}.



Exemplo III: *Representação das ligações a serem feitas de uma célula V_{ij} de conteúdo "C" com adjacentes W_{ij} sendo i e j coordenadas de V_{ij}.*

Caminhamento e Análise

Após a construção do grafo, todas as operações de análise e caminhamento são realizadas diretamente sobre ele. O problema pode ser descrito como a busca por componentes conectados na estrutura de grafo:

- Para cada vértice ainda não visitado, realiza-se um caminhamento em largura (BFS), que consiste em explorar todos os vértices vizinhos de forma sistemática, nível por nível, utilizando uma fila para controlar a ordem de visita.

- Durante o percurso, são contados os personagens presentes no componente, e ao final de cada caminhamento, identifica-se qual personagem aparece com maior frequência.

O controle de vértices visitados assegura a eficiência do algoritmo, garantindo que o percurso abranja toda a estrutura sem redundâncias. Sempre que um vértice não marcado é encontrado, identifica-se um novo componente conectado, pois sua ausência no conjunto de vértices visitados indica que ele não está associado a nenhum componente previamente explorado.

Integração com a Matriz M

Embora a matriz M seja fundamental para a construção inicial do grafo, todas as operações de caminhamento e análise ocorrem diretamente sobre o grafo. Isso garante maior flexibilidade e eficiência no processamento dos dados do labirinto.

IV – Estrutura do Algoritmo

Após abstrairmos o problema proposto e definirmos uma estrutura capaz de resolvê-lo, podemos iniciar o desenvolvimento de um algoritmo eficiente. A estrutura do problema foi definida da seguinte maneira:

- **`matriz Lab`**: Armazena as informações de entrada, servindo como base para a construção da estrutura necessária à resolução do problema.
- **`class Graph`**: Estrutura de dados utilizada para armazenar os vértices e seus adjacentes, sendo uma representação eficiente das conexões do problema.
- **`list Marked`**: Estrutura para registrar os nodos já visitados, garantindo o controle e a organização dos vértices marcados na estrutura **`Graph`**.
- **`class Node`**: Objeto que representa os vértices, armazenando as coordenadas de sua posição e o conteúdo da célula, definindo os adjacentes laterais com os quais o vértice deve se conectar.
- **`procedimento BFS`**: Algoritmo de caminhamento que assegura a visitação de todos os nodos, verificando adequadamente se eles já foram marcados ou não.
- **`int Componentes`**: Variável auxiliar para contabilizar o número de componentes presentes no labirinto, permitindo a análise da conectividade entre os nodos.
- **`int Maior`**: Variável auxiliar para registrar o maior número de personagens encontrado em um único componente.
- **`char MaisFrequente`**: Variável para registrar qual personagem se encontra em mais aparições nos componentes.

V – Síntese do Algoritmo

Com base nos tópicos anteriores, projetamos o algoritmo para realizar as seguintes tarefas:

1. **Definir Nodos**: Cada nodo é configurado para armazenar informações relevantes, como sua posição na matriz e o conteúdo da célula, o que auxiliará na construção das ligações entre os nodos, além de permitir a análise das interações e conexões no grafo.

2. **Definir Grafo**: Implementa-se uma estrutura de grafo para representar as conexões do problema, onde cada vértice contém seus adjacentes, facilitando a modelagem das relações entre os pontos do labirinto.

3. **Realizar Ligações**: Dentro da estrutura do grafo, verifica-se as células do labirinto e suas propriedades, realizando as ligações entre os nodos conforme as informações da matriz, considerando as conexões válidas entre eles.

4. **Gerar Matriz**: Cria-se uma matriz de nodos com base nos dados de entrada fornecidos, sendo utilizada para construir a estrutura grafo.

5. **Caminhamento BFS**: Aplica-se o algoritmo de busca em largura (BFS) para explorar todos os nodos do grafo, contabilizando componentes e personagens. O BFS utiliza uma fila para armazenar os vértices a serem explorados, visitando os vizinhos de cada vértice antes de avançar para os próximos, garantindo que todos os vértices conectados sejam visitados de forma ordenada.

Esse fluxo assegura que o algoritmo identifique com precisão o número total de componentes conectados e determine o maior número de personagens presentes em um único componente.

VI – Aplicação

Primeiramente, precisamos abstrair a estrutura de nodos e grafo, para então partir para a criação da nossa matriz. A matriz servirá como base para representar os nodos, que posteriormente serão conectados para formar o grafo e permitir a exploração das regiões no labirinto.

```
1  class Node:
2
3  membros privados:
4
5      int x;//coordenada x na matriz
6      int y;//coordenada y na matriz
7
8      char info; //contéudo da célula
9
10 membros públicos:
11
```

```
12    //Construtor
13    Node(int x, int y, char info): x(x), y(y), info(info);
14
15    Getters e Setters;
16
17 fim class
```

***Legenda**: Tarefa do Tópico V: [1] Definir Nodos.*

```
1  class Graph:
2
3  membros privados:
4
5     //lista que relaciona nodos com seus adjacentes
6     map <Node, vector<Node>> graph;
7
8     //lista que armazena nodos já percorridos
9     set <Node> markedNodes;
10
11 membros públicos:
12
13    //Construtor - Cria Vértices e relaciona com seus Adjacentes
14    Graph(int linha, int coluna, vector<vector<Node>> Lab):
15
16    para i ⟵ 0 até linha faça:
18          para j ⟵ 0 até coluna faça:
19
17                char info ⟵ Lab[i][j].getInfo();//conteúdo da célula
18
19                int val ⟵ info;//valor convertido para hexadecimal
20
21                //verifica se S3 de val é 0
22                se S3 ⟵ 0 e i-1 maior ou igual a 0 faça:
23                      conecte Lab[i][j] com  Lab[i-1][j];
24
25                //verifica se S2 de val é 0
26                se S2 ⟵ 0 e j+1 maior ou igual a 0 faça:
27                      conecte Lab[i][j] com  Lab[i][j+1];
28
29                //verifica se S1 de val é 0
30                se S1 ⟵ 0 e i+1 maior ou igual a 0 faça:
31                      conecte Lab[i][j] com  Lab[i+1][j];
32
33                //verifica se S0 de val é 0
34                se S0 ⟵ 0 e j-1 maior ou igual a 0 faça:
35                      conecte Lab[i][j] com  Lab[i][j-1];
36          fim para
37    fim para
38
39    procedimento conecte Node V com Node W:
40
41          //adiciona Node v ao grafo e retorna seus adjacentes
42          vector<Node> aux ⟵ graph[v];
43
44          aux ⟵ Node W//Node W adicionado como adjacente de Node V
45
46    fim procedimento
```

```
47
48    Getters e Setters;
49
50 fim class
```

***Legenda**: Tarefas do Tópico V: [2] Definir Grafos;*
[3]Realizar Ligações;

Agora que somos capazes de armazenar nossos elementos em estruturas especializadas, podemos montar a matriz com os nodos, gerar a estrutura de grafo correspondente e, por fim, realizar o caminhamento necessário para explorar o labirinto e encontrar as soluções propostas.

```
1  procedimento criaMatriz(int linha, int coluna, Node Lab[][]):
2
3     char aux;//auxilia na criação do nodo
4
5     para i ⟵ 0 até linha faça:
6           para j ⟵ 0 até coluna faça:
7                 // aux lido de input.file
8                 Lab[i][j] ⟵ Node(i, j, aux);//adicionado um nodo
9           fim para
10    fim para
11
12 fim procedimento
```

***Legenda**: Tarefa do Tópico V: [4] Gerar Matriz.*

Com a matriz de nodos pronta, podemos proceder com a implementação do caminhamento em largura (BFS) sobre o nosso grafo, a fim de explorar as conexões entre os nodos e identificar as regiões isoladas no labirinto. O BFS garantirá que cada componente conectado seja percorrido de forma eficiente, permitindo a contagem dos personagens e a análise das regiões.

```
1  procedimento BFS(Graph G, Node V, char MaisFrequente, int Maior):
2
3     //array para contabilizar personagens presentes no componente
4     vector<int> ContaPersonagens vai de A até F
5
6     //responsável por armazenar adjacentes a serem percorridos
7     queue<Node> fila;
8
9     fila ⟵ V;//insere nodo primário na fila
10    G.setMarked ⟵ V;//marca V como percorrido na estrutura grafo
11
12    enquanto fila diferente de vazio, faça:
13          Node aux ⟵ 1° elemento de fila
14          remove 1° elemento de fila;
15
16          para cada elemento W de G.getAdjacentes ⟵ V faça:
17                se ~G.getMarked ⟵ W faça:
18                      G.setMarked ⟵ W;
19                      fila ⟵ W;
20                fim se
21          fim para
22
```

```
23          //aproveita para contabilizar personagem do componente
24          escolha – aux.getInfo():
25                caso 'A': ContaPersonagens[A]++
26                caso 'B': ContaPersonagens[B]++
27                caso 'C': ContaPersonagens[C]++
28                caso 'D': ContaPersonagens[D]++
29                caso 'E': ContaPersonagens[E]++
30                caso 'F': ContaPersonagens[F]++
31          fim escolha
32    fim enquanto
33
34
35    para i ⟵ A até F faça:
36          se ContaPersonagem[i] maior Maior faça:
37                maior ⟵ ContaPersonagem[i];
38                MaisFrequente ⟵ i;
39          fim se
40    fim para
41
42 fim procedimento
```

***Legenda**: Tarefa do Tópico V: [5] Caminhamento BFS*

Por fim, temos o **procedimento** `main`, responsável por executar o algoritmo e solucionar o problema estabelecido.

```
1  procedimento main():
2
3     int linha, coluna, Maior, Componente;
4     char MaisFrequente;
5
6     linha ⟵ lê do terminal;
7     coluna ⟵ lê do terminal;
8
9     vector<vector<Node>> Lab;//matriz que armazenara dados
10
11    procedimento criaMatriz(linha, coluna, Lab);
12    Graph G(linha, coluna, Lab);//Cria um grafo G
13
14    para cada elemento V de G.getVertices, faça:
15          se ~G.getMarked ⟵ V faça:
16                procedimento BFS(G, V, MaiorFrequencia, Maior);
17                Componente++;
18          fim se
19    fim para
20
21    saida ⟵ Componente & MaisFrequente & Maior;
22
23 fim procedimento main
```

***Legenda**: Procedimento Main: Executa a projeção do algoritmo elaborado*

VII – Análises e Resultados

Para iniciar a análise, foram estabelecidos dois critérios para avaliar o algoritmo: 1) **Número de operações em função do tamanho do grafo**, que avalia o algoritmo considerando a entrada de tamanho n e suas conexões internas. 2) **Tempo em função do número de operações**, que avalia o tempo de execução do algoritmo com base nas

operações realizadas durante o processamento. Os resultados foram obtidos a partir de testes realizados em labirintos com oito dimensões diferentes: 40x40, 80x80, 100x100, 120x120, 150x150, 180x180, 200x200 e 250x250. A ***Tabela I*** apresenta os resultados desses testes, agrupados de acordo com as diferentes dimensões analisadas.

Caso	Componentes	Personagem	Freq.	Op.	Vért.	Arestas	Tempo (s)
40x40	27	A	4	3173	1600	1573	0.006429
80x80	82	C	8	12718	6400	6318	0.034425
100x100	26	A	21	19974	10000	9974	0.050752
120x120	86	E	12	28714	14400	14314	0.073941
150x150	48	A	23	44952	22500	22452	0.117981
180x180	176	A	17	64624	32400	32224	0.173544
200x200	60	E	40	79940	40000	39940	0.217778
250x250	39	C	81	124961	62500	62461	0.334351

Tabela I: *Expõe números obtidos através de testes experimentais*

Análise I

A primeira análise considera o número de operações realizadas pelo algoritmo em função do tamanho n do labirinto. Esse critério busca avaliar o custo computacional do algoritmo ao lidar com labirintos de diferentes dimensões, variando desde estruturas pequenas até configurações de tamanho significativamente maior. A intenção é determinar como o crescimento do tamanho do labirinto impacta a eficiência do algoritmo, oferecendo uma perspectiva clara sobre sua escalabilidade e desempenho em cenários com diferentes volumes de dados. Após a realização dos testes, os resultados foram compilados e aplicados ao ***Gráfico 1***.

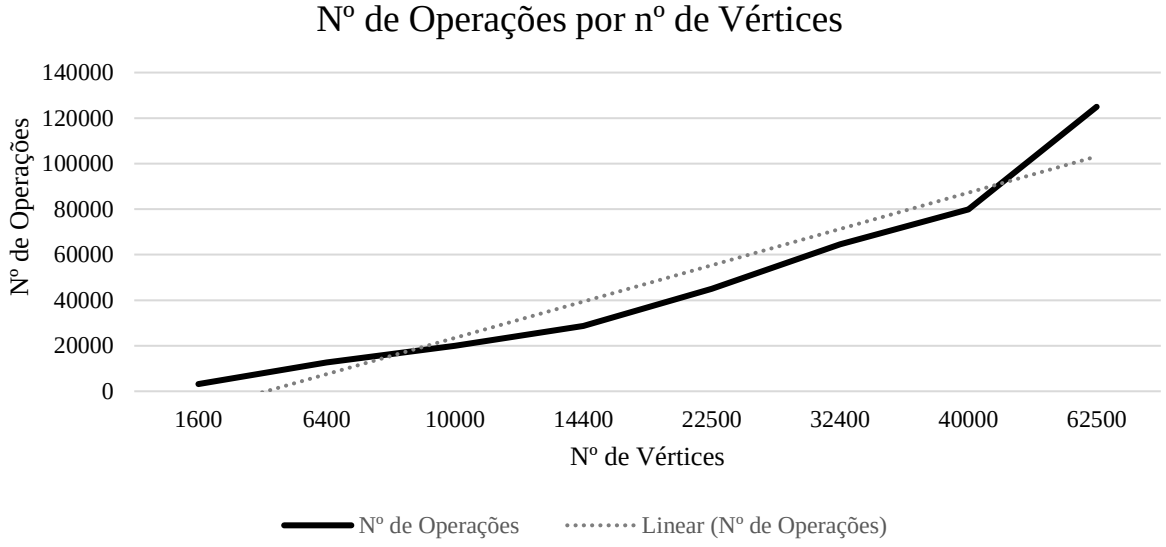


Gráfico 1 *– Relação entre o Número de Operações com o Número de Vértices*

Em uma primeira observação, é possível notar que o número total de operações realizadas pelo algoritmo corresponde à soma: $Operações = n\,Vértices + n\,Arestas$. É importante destacar que tanto o número de vértices quanto o de arestas foram obtidos a partir da estrutura de grafo, que disponibiliza esses índices de maneira eficiente. A contabilização das operações foi realizada em pontos estratégicos do algoritmo, garantindo precisão na análise.

Essa relação é corroborada pela lógica do algoritmo, uma vez que, para cada vértice, é necessário percorrer todos os seus adjacentes, o que depende diretamente do número de arestas conectadas. Além disso, em um algoritmo bem projetado, com controle adequado de nós marcados, cada nó deve ser visitado no máximo uma vez, assim como seus adjacentes. Dessa forma, o número de operações relacionadas ao processamento de nodos e arestas é limitado, sendo no máximo proporcional ao número total de arestas. Essa característica reforça a eficiência e a coerência da implementação em relação ao modelo teórico proposto.

Análise II

A segunda análise examina o tempo de execução do algoritmo em função do número de operações realizadas, avaliando sua eficiência prática em termos de velocidade e desempenho. Como o número de operações é proporcional ao total de vértices e arestas, essa abordagem também reflete indiretamente o tempo de execução em função do tamanho do grafo. Assim, é possível verificar se o algoritmo mantém um desempenho consistente tanto para estruturas menores quanto para maiores, confirmando se a eficiência teórica é condizente com sua aplicação prática.

Ao relacionar o tempo ao número de operações, espera-se observar uma proporcionalidade direta, especialmente em implementações otimizadas. Após a realização dos testes, os resultados foram compilados e aplicados ao ***Gráfico 2***.

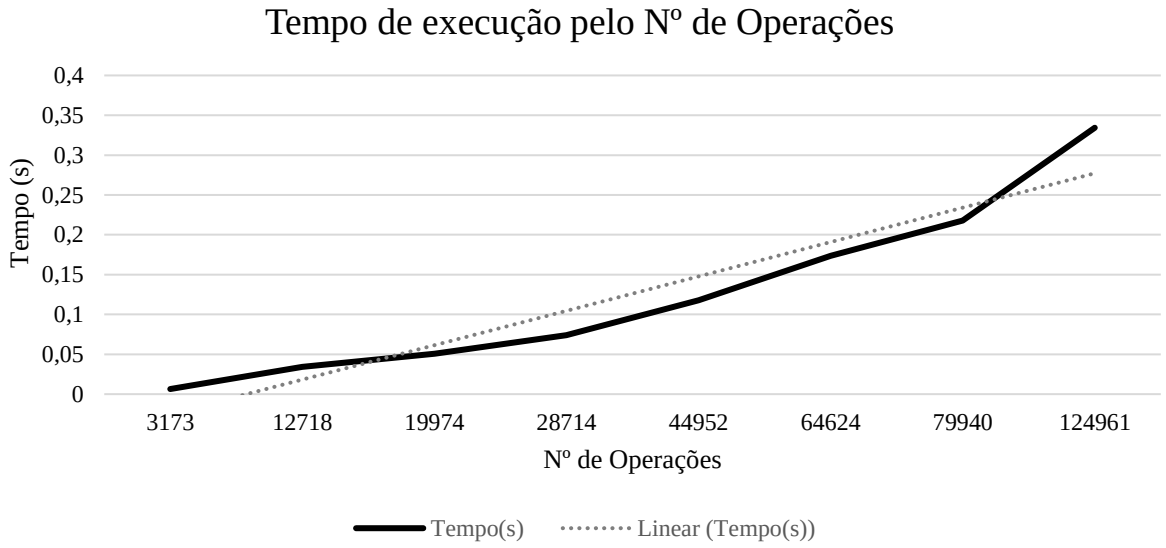


Gráfico 2 *– Relação entre o Tempo de execução (em segundos) com o Nº de Operações necessárias*

Através do gráfico obtido, observa-se uma tendência linear, evidenciando a eficiência do algoritmo mesmo em cenários com um número elevado de operações. Além disso, destaca-se o tempo reduzido de execução, mesmo para estruturas grandes. É importante ressaltar que o tempo medido engloba tanto a fase de construção do grafo — que inclui a criação das arestas e ligação dos vértices — quanto o caminhamento em si.

No ***Gráfico 2***, a eficiência do algoritmo é ainda mais evidente, considerando o desempenho tanto na fase de ligação dos vértices quanto na execução do caminhamento sobre a estrutura. Essa tendência linear, semelhante à do ***Gráfico 1***, reforça que o comportamento do algoritmo está alinhado com a complexidade teórica $O(V + E)$.

VIII – Conclusão

Após as análises realizadas, concluímos que o algoritmo apresentado possui um comportamento linear, com complexidade da ordem de $O(V + A)$, onde V representa o número de vértices e A, o número de arestas do grafo. Essa eficiência decorre de dois fatores principais: o controle rigoroso de marcação dos nodos, que garante que cada vértice seja visitado no máximo uma vez, e a forma como as operações de percorrimento dependem diretamente do número de arestas conectadas. Essa característica demonstra que o algoritmo lida de maneira eficiente tanto com grafos esparsos quanto com grafos densos.

A abstração do algoritmo foi comprovada ao constatar que seu comportamento segue exatamente a complexidade $O(V + A)$. Essa relação é consistente com a necessidade de visitar cada vértice uma única vez e processar todas as arestas adjacentes, garantindo que o custo do algoritmo cresça proporcionalmente ao tamanho do grafo. Essa comprovação experimental sustenta a solidez teórica do modelo implementado.

Os gráficos de operações e tempo reforçam essa conclusão, ao evidenciar uma clara tendência linear entre o número de operações realizadas e o tamanho do grafo. Mesmo em cenários com alta densidade de arestas, o tempo de execução permaneceu proporcional à soma do número de vértices e arestas. Além disso, a contabilização do tempo total incluiu tanto a fase de construção do grafo — envolvendo a criação das arestas e ligações entre vértices — quanto a execução do caminhamento BFS propriamente dito. Essa abordagem abrangente válida a robustez do algoritmo em diferentes etapas, destacando sua eficiência geral.

Em termos teóricos, o comportamento observado está de acordo com os preceitos fundamentais da busca em largura (BFS), uma técnica amplamente reconhecida por sua eficiência em grafos. Experimentalmente, a implementação foi testada com sucesso em grafos de diferentes tamanhos e densidades, confirmando sua adequação para cenários variados e complexos. Adicionalmente, a proposta do algoritmo mostrou-se eficiente para resolver o problema específico apresentado neste artigo, atingindo os objetivos definidos com precisão e agilidade.

"*Algoritmos de grafos são como magia: transformam conexões abstratas em soluções concretas para problemas do mundo real.*"
— Inspirado em *The Algorithm Design Manual*, Steven Skiena, 2ª edição, Springer, 2008.