Trabalho II de Algoritmos e Estruturas de Dados II

Arthur Sudbrack Ibarra[*], Willian Magnum Albeche[†]
Engenharia de Software — PUCRS

21 de junho de 2021

Resumo

Este artigo descreve a solução encontrada para o segundo trabalho da disciplina de Algoritmos e Estrutura de Dados II, no semestre 2021/1. O problema em questão se tratava da criação de um algoritmo, o qual, tendo em vista um labirinto e seus caminhos, deveria permitir com que descobríssemos o menor número de movimentos necessários para chegar à saída (ponto S), respeitando, contudo, algumas regras estipuladas para o jogo. A seguir, será apresentada a solução para a problemática em conjunto com a análise da eficiência deste algoritmo, e, posteriormente, a apresentação dos casos de testes propostos pelo professor.

Introdução

Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados II, o problema proposto sugere que, dado um labirinto cilíndrico, isto é, com forma retangular, porém o lado direito faz conexão com o lado esquerdo e vice-versa, deveríamos criar um algoritmo capaz de descobrir o menor número de movimentos para levar um "carneiro"do ponto **C** até o ponto **S** (se isso for possível), levando em conta sua movimentação, a qual pode ser feita tanto somente no eixo X e Y (para os lados, para cima e para baixo) quanto também para as diagonais. Contudo, a complexidade do problema aumenta uma vez que dentro do labirinto há "buracos"marcados com **x**, dos quais o personagem não pode pisar em cima, sendo necessário realizar a ação de pular por cima do obstáculo. Por fim, vale ressaltar que serão representados por um ponto (".") todos os caminhos livres de buracos. Assim, tendo em vista a problemática que nos foi apresentada, foi-se pedido que determinássemos o menor número de movimentos para chegar ao final do labirinto e que mostrássemos a sequência destes movimentos de acordo com os diferentes tabuleiros (ou universos) que serão testados.

[*]arthur.ibarra@edu.pucrs.br
[†]Willian.Albeche@edu.pucrs.br

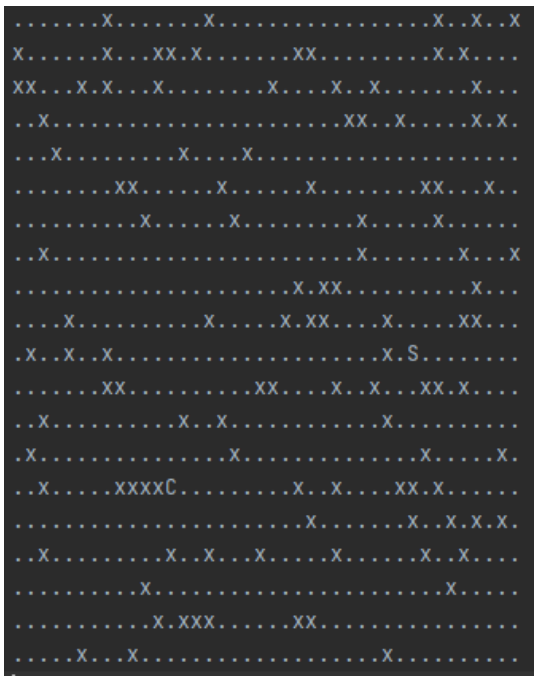

Figura 1: Representação do labirinto

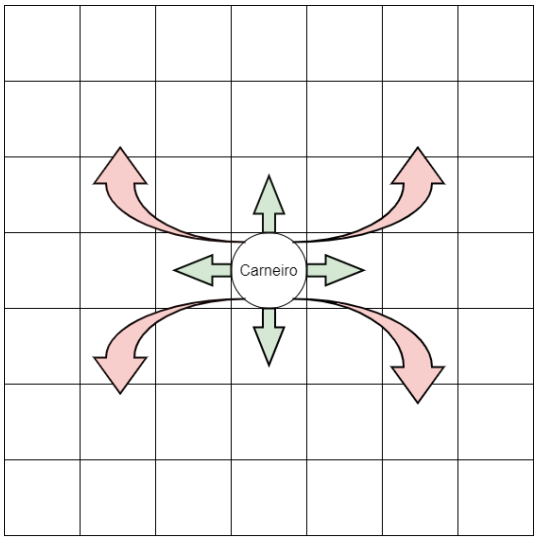

Figura 2: Movimentação do carneiro

Primeira solução

Após a realização da análise do problema em questão, foi-se decidido começar a estruturação do código base, através da criação da classe responsável por gerenciar o labirinto e também os movimentos possíveis que nosso personagem pode realizar nele. Com isso, foi possível visualizar com maior clareza as regras de movimentação aplicadas no labirinto, possibilitando assim a escolha do algoritmo que seria utilizado para a solução do problema.

A classe criada para abstrair o labirinto chama-se *Labyrinth* e possui dentro desta uma matriz de objetos *Block*, sendo que cada bloco é composto por um símbolo (., x, C ou S), por uma linha, por uma coluna e por uma lista de outros blocos que podem ser acessados a partir dele, conforme consta na figura abaixo:

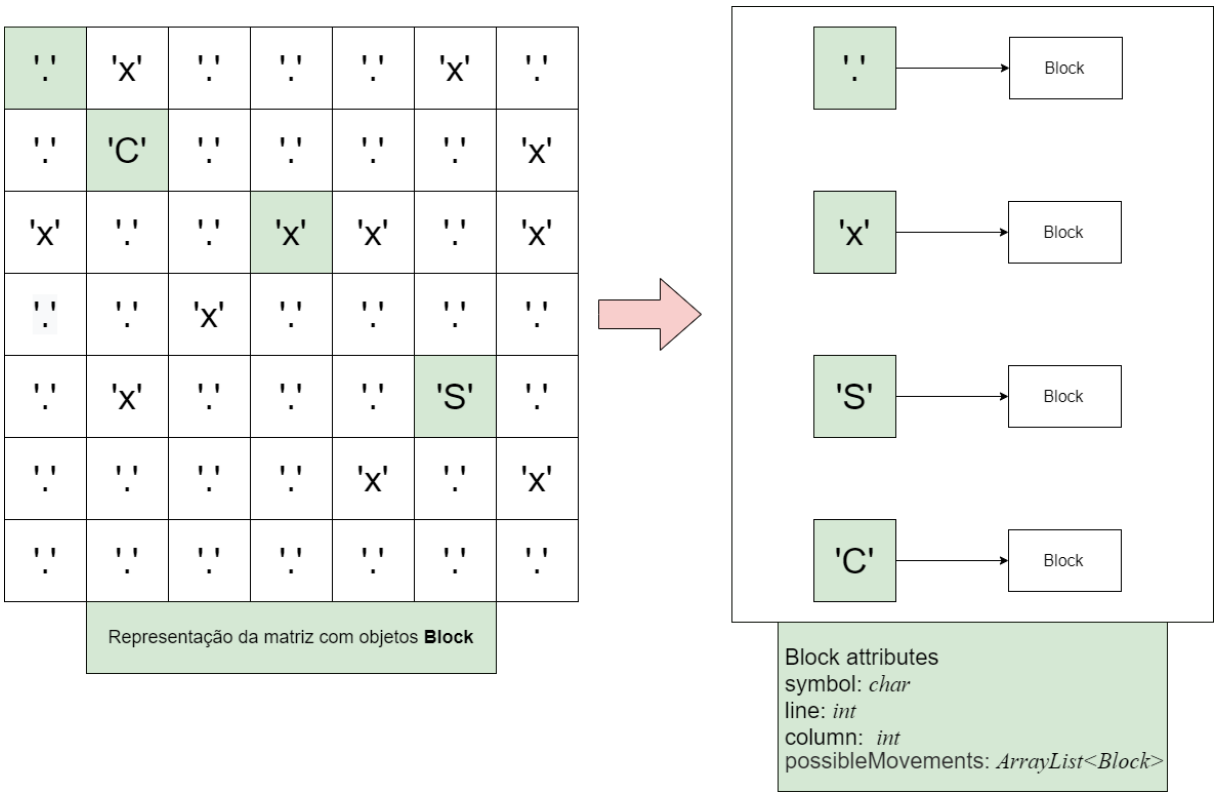

Figura 3: Representação da matriz, onde cada símbolo é um atributo de Block

1 Descrição do Algoritmo

Com relação ao algoritmo escolhido para de fato determinar o caminho mais curto do bloco **C** até o bloco **S**, utilizamos um *BFS Graph* (Breadth-first search ou Grafo de busca em largura)[1] de arestas mínimas, para isso, no entanto, é necessário definir nossos vértices e arestas que compõem o grafo. Nesse sentido, foi-se determinado que cada vértice do grafo seria representado por um número inteiro começando em 0 e indo até *(linhas * colunas) - 1*. Como estamos trabalhando com objetos *Block* e precisamos de vértices em números inteiros, foi-se também utilizado um dicionário para transformar um *Block* em um *int* com base em uma função hash[2] que leva em conta a linha e a coluna que o bloco se encontra na matriz. Conforme veremos abaixo, as arestas do grafo são montadas de acordo com cada movimento possível a partir de cada bloco:

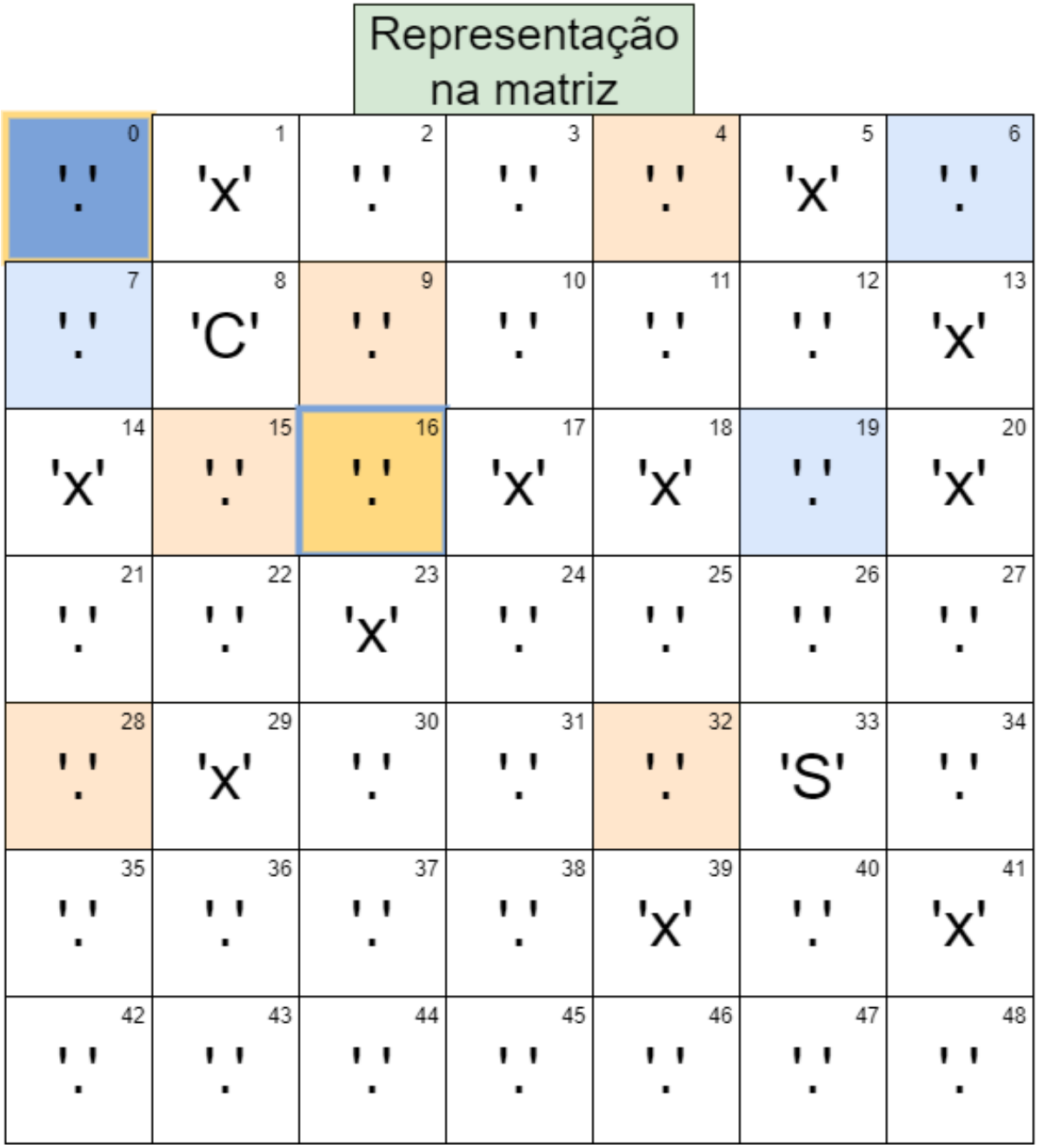

Figura 4: Representação da matriz com base na possibilidade de movimentação.

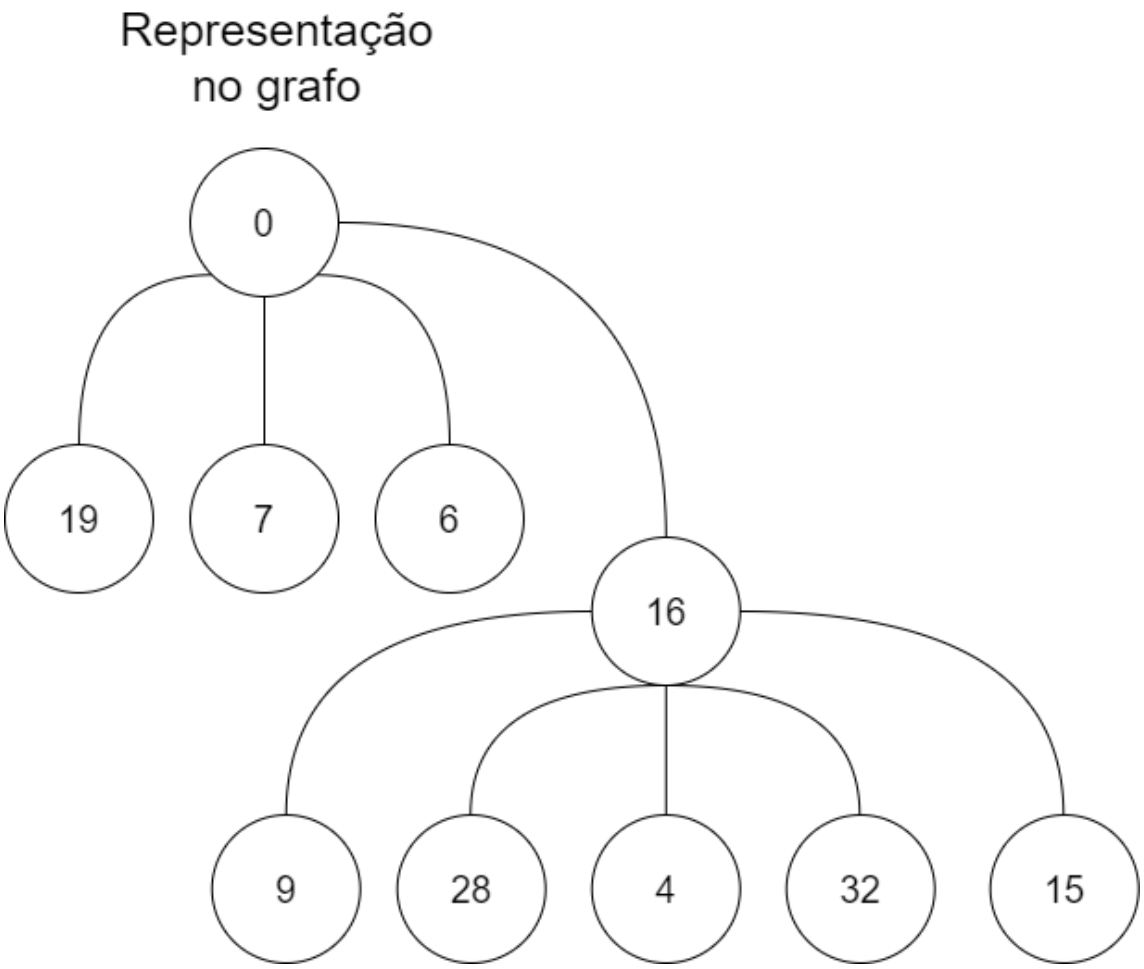

Figura 5: Representação do grafo com base na matriz anterior

Todo o fluxo que permite a representação da matriz (labirinto) em um grafo pode ser compreendido com maior clareza por meio da imagem abaixo:

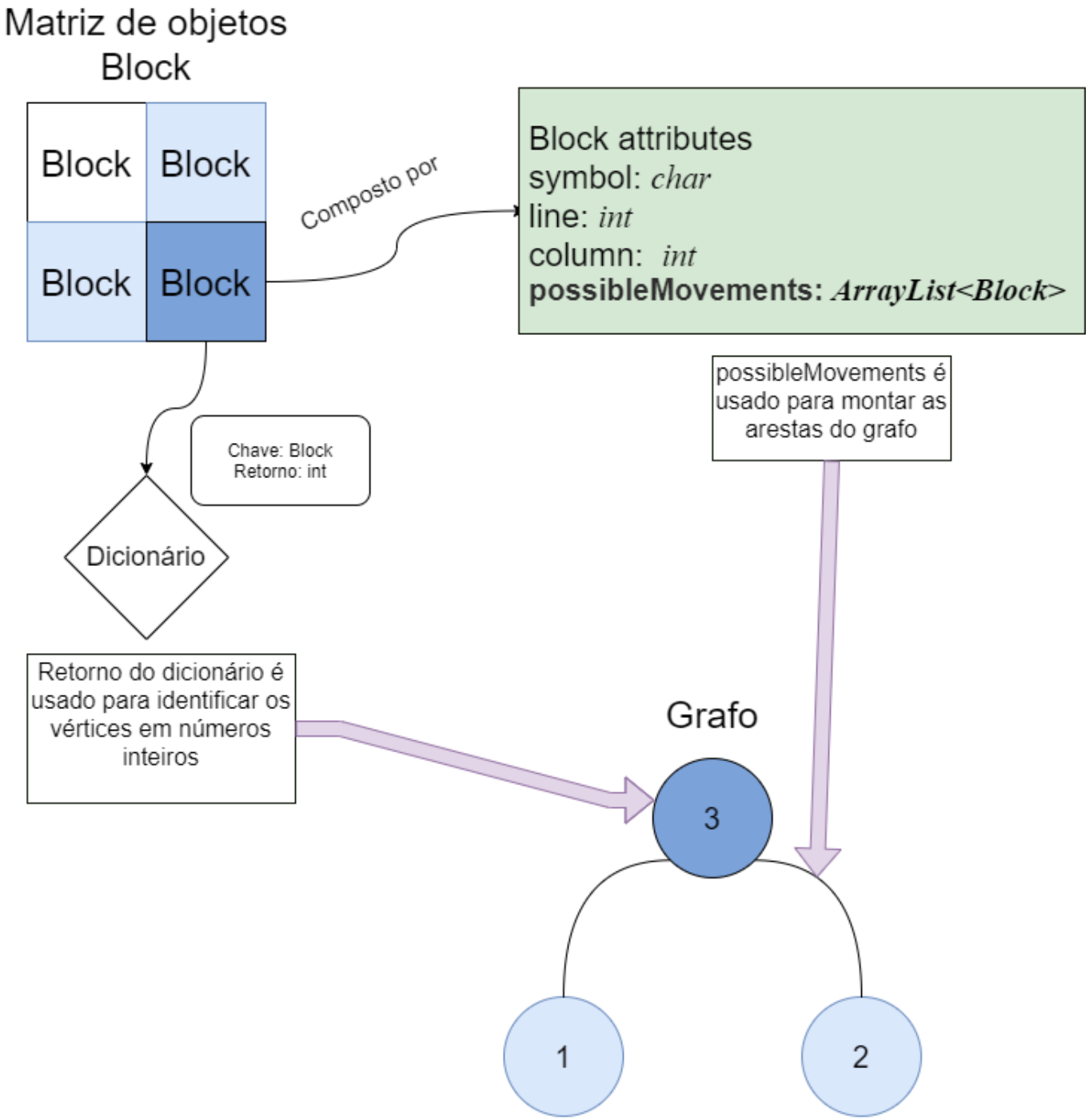

Figura 6: Representação do fluxo matriz-grafo

Podemos concluir, portanto, que a estrutura de dados com a qual estamos trabalhando se trata de um grafo:

- Não-direcionado, pois a relação entre blocos (vértices) sempre se dá nos dois sentidos (bidirecional).
- Não-valorado, pois para a situação problema não faz sentido as arestas assumirem valores.
- Cíclico, pois existem situações em que é possível sair de um bloco (vértice) e chegar e ele novamente fazendo outro caminho.

Dessa maneira, foi levando em conta as propriedades acima que o grupo decidiu aplicar o algoritmo de *BFS* anteriormente mencionado, uma vez que, por o grafo não ser valorado, não desejamos saber o caminho com as menores arestas, mas sim aquele com o menor número de arestas de um ponto específico até outro, e o caminhamento em largura (Breadth-first search) de arestas mínimas, é ideal para esse propósito.

1.1 Caminhamento no grafo - BFS (Breadth-first search)

Um grafo BFS é uma estrutura de dados, cujo algoritmo de caminhamento usa uma metodologia de busca não-informada, a qual examina todos os vértices de um grafo, podendo ele ser direcionado ou não-direcionado. Em suma, podemos dizer que o algoritmo irá percorrer exaustivamente todas as arestas e visitar todos os vértices do grafo, garantindo que nenhum destes últimos serão visitados mais de uma vez, sendo assim, para realizar tal ação, é utilizada a estrutura de dados de fila (Queue). Note que o tipo de grafo em questão se assemelha a uma árvore, pois começa o percorrimento a partir de um vértice 'raiz' escolhido, e segue para seus vizinhos, acessando os vértices inexplorados e assim por diante.

No caso de nosso labirinto, o algoritmo de BFS é finalizado quando encontramos o caminho mais curto (ou seja, com menos arestas) de qualquer vértice para qualquer outro. Com isso em mente, basta indicarmos que queremos sair do bloco **C** e chegar até o bloco **S** que o algoritmo irá retornar uma lista de tamanho *n* contendo todos os vértices que precisam ser acessados para chegarmos ao destino informado.

1.1.1 Pseudo-código

Abaixo segue o pseudo-código do algoritmo de caminhamento em questão:

```
1  procedimento BFSMinimoArestas (verticeSaida , verticeDestino) é:
2  sendo Q uma fila
3  verticeSaida.veioDe < −verticeSaida
4  verticeSaida.visitado < −true
5  Q.enqueue(verticeSaida)
6  enquanto Q não estiver vazia , faça:
7          x < −Q.dequeue()
8          para cada aresta y de x , faça:
9                  se y.visitado , faça:
10                         continuar
11                 y.veioDe < −x
12                 Q.enqueue(y)
13                 y.visitado < −true
14 sendo L uma lista
15 verticeAtual < −verticeDestino.veioDe
```

```
16 se verticeAtual == −1 , então:
17         retorna L
18 L.add(0 , verticeAtual) //Adiciona na primeira posição da lista.
19 enquanto verticeAtual ≠ verticeSaida , faça:
20         verticeAtual = verticeAtual.veioDe
21         L.add(0 , verticeAtual)
22 L.add(0 , verticeAtual)
23 retorna L
```

1.1.2 Ordem de complexidade

Em relação à ordem de complexidade do BFS, sabe-se que esta é linear e pode ser definida por O(V + E), sendo que V representa os vértices e E as arestas, uma vez que todos os vértices e arestas serão explorados. A eficiência do algoritmo em questão pôde ser vista e comprovada empiricamente ao contarmos o tempo de execução em milissegundos (ms) e a quantidade de operações realizadas dentro do programa desenvolvido pelo grupo. Nesse sentido, os gráficos abaixo fazem uso dessas informações para representar visualmente o crescimento do tempo de execução e da quantidade de operações em função da quantidade de caracteres (blocos do labirinto):

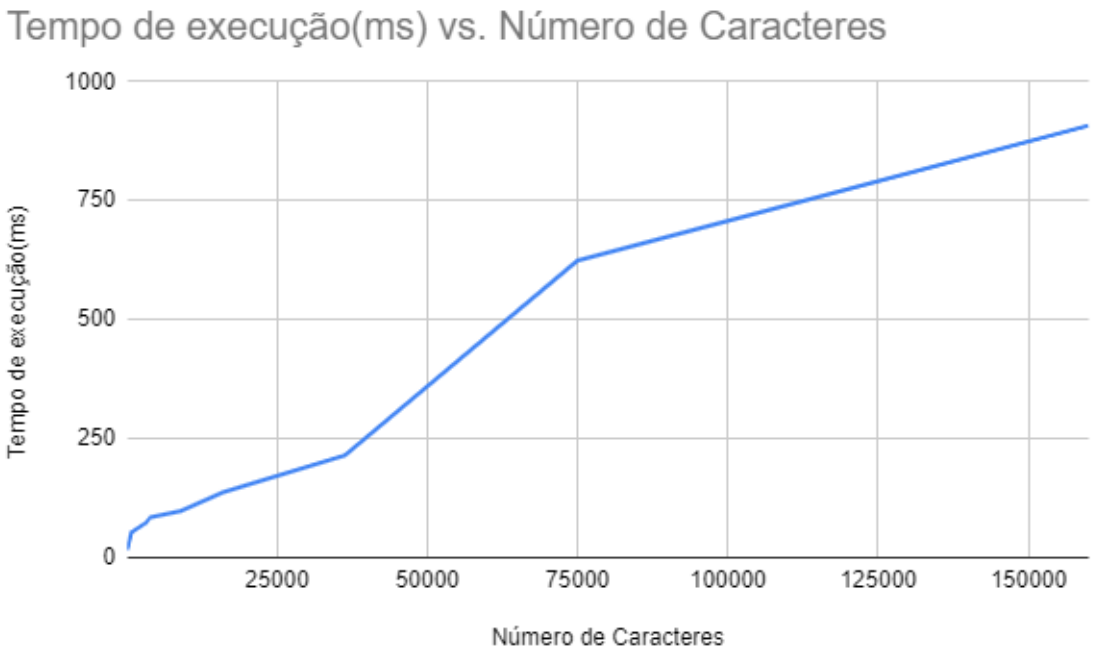

Figura 7: Gráfico tempo de execução x número de caracteres

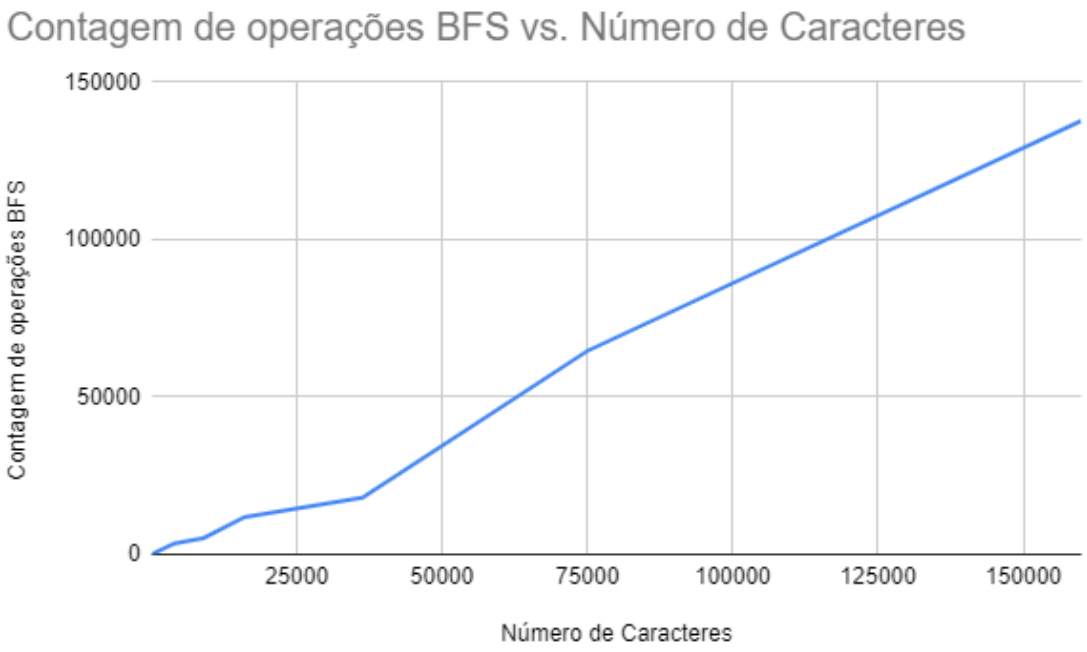

Figura 8: Gráfico contagem de operações x número de caracteres

2 Resultados obtidos

A seguir estão sendo apresentados os resultados obtidos para cada caso de teste fornecido pelo professor. Note que cada caso de teste se classifica como um arquivo .txt, contendo a representação visual do labirinto, e os números presentes no nome do arquivo representam a quantidade de colunas que os caracteres dentro do arquivo formam. A seção *Blocos* da tabela, por sua vez, indica a quantidade de blocos que devem obrigatoriamente ser percorridos no caminho mais curto de **C** até a saída **S**.

Casos de teste	Blocos
caso040.txt	4
caso040.txt	12
caso080.txt	15
caso100.txt	22
caso150.txt	Sem caminho
caso200.txt	77
caso300.txt	Sem caminho
caso500.txt	78
caso800.txt	201

Figura 9: Tabela dos resultados obtidos

3 Conclusão

Ao final do trabalho, o grupo encontra-se satisfeito com o desempenho da solução encontrada, a qual, mesmo com casos de testes maiores, possui um tempo de execução muito satisfatório, sendo menos de 1 segundo no maior caso de teste (descartando o tempo para exibir a matriz no terminal). O algoritmo de caminhamento em grafo possuiu uma ordem de complexidade linear O(V + E), apresentando uma boa eficiência na resolução do problema, não fazendo-se necessário a tentativa de implementar uma segunda solução. Em virtude do que foi dito, concluímos que o projeto como um todo cumpre com os requisitos solicitados para a resolução do problema, propondo assim, um programa eficiente e bem estruturado.

Referências

[1] GeeksforGeeks: "**Breadth First Search or BFS for a Graph**". Disponível em: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> Acesso em: 19 de Jun. 2021.

[2] Java Documentation: "**Class HashMap<K,V>** ". Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html> Acesso em: 18 de Jun. 2021.

[3] Java Documentation: "**Package java.io**". Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html> Acesso em: 16 de jun. 2021.

[4] GeeksforGeeks: "**Minimum number of edges between two vertices of a Graph**". Disponível em: <https://www.geeksforgeeks.org/minimum-number-of-edges-between-two-vertices-of-a-graph/> Acesso em: 20 de jun. 2021.