Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Ajudando um rato a sair do labirinto

BCC202 - Estrutura de dados I

Eduardo Silva & Niege Reis

Professor: Pedro Silva

Ouro Preto
13 de agosto de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

No presente relatório, serão detalhados os passos envolvidos no desenvolvimento do terceiro trabalho prático referente à disciplina de Estrutura de Dados I (BCC202), parte integrante do currículo do curso de Ciência da Computação da UFOP (Universidade Federal de Ouro Preto).

1.1 Especificações do problema

O desafio proposto consistiu no mapeamento do labirinto em uma árvore, e então percorrer a mesma para determinar o menor e o maior caminho.

1.2 Considerações iniciais

Para o desenvolvimento da lógica, foram utilizadas soluções previamente conhecidas, as quais já foram estabelecidas e reconhecidas como eficientes no contexto do problema proposto.

- Busca em Largura (Breadth First Search ou BFS)
- Busca em Profundidade (Deapth First Search ou DFS)

No contexto deste trabalho, foram utilizados algoritmos de busca que são amplamente aplicados para percorrer estruturas como árvores e grafos. Considerando a natureza do problema do labirinto, foi possível estabelecer uma relação entre o labirinto e uma árvore, em que cada posição (X, Y) do labirinto representava um nó na estrutura.

1.3 Ferramentas e bibliotecas utilizadas

Algumas ferramentas foram utilizadas para desenvolver e testar a implementação, como:

- *GCC*: Ferramentas de análise estática do código.
- *Valgrind*: Ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80GHz
- Memória RAM: 24Gb.
- Sistema Operacional: Windows 11 (WSL 2).

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta executar as seguintes instruções (dentro da pasta do projeto): Ainda possui alguns casos de testes dentro da pasta test-cases.

Compilando o projeto

```
make
./exe < entrada.in
```

2 Implementação

Iniciamos o processo de desenvolvimento do trabalho analisando quais as TADs (Tipos Abstratos de Dados) poderiamos reutilizar, em seguida, implementamos as demais TADs necessárias, juntamente com suas respectivas funções. Posteriormente, implementamos a principal TAD que desempenha o papel fundamental no mapeamento do labirinto na arvore, e na busca pela saida do labirinto.

2.1 Tipos abstratos de dados (TADS)

Implementamos seguintes TADS:

- Position (Responsável por armazenar e manipular uma cordenada (X,Y) e responsável por pegar as direções).
- LinkedList & LinkedListNode (Implementação de uma lista ligada, a LinkedListNode é a celula da lista ligada)
- Queue (Implementação da fila (FIFO))
- Maze (Responsável por armazenar e manipular os dados de um labirinto, como quantidade de linhas, colunas e o tabuleiro)
- Path (Responsável por armazenar o caminho percorrido)
- Tree (Responsável por armazenar a árvore, mapear o labirinto e encontrar as saidas)

2.1.1 Position (ou Posição)

Decidimos trabalhar com a TAD de posição de forma estática, pois isso simplificaria a manipulação dos dados.

```
#ifndef POSITION_H
#define POSITION_H

#include<...>

typedef struct position {
  int x;
  int y;
} Position;
typedef enum { UP, DOWN, LEFT, RIGHT } Direction;

Position PositionCreate(int x, int y);
void PositionPrint(Position position);
Direction PositionToDirection(Position position, Position origin);

#endif  // POSITION_H
```

Código 1: Implementação do TAD de posição.

2.1.2 LinkedList (ou Lista)

Durante o desenvolvimento do trabalho, foi necessário implementar uma lista, e optamos pela implementação da lista encadeada (ou lista ligada) devido às suas vantagens em relação à lista sequencial, especificamente para o nosso problema. Algumas vantagens notáveis dessa escolha incluem:

- Um dos benefícios da escolha da lista encadeada é a capacidade de lidar com a falta de conhecimento sobre a quantidade de passos em um caminho. Como não temos certeza de quantos passos um labirinto pode ter, a lista encadeada se mostra vantajosa, pois não precisamos alocar um espaço fixo para todos os possíveis passos (diferentemente da lista sequencial). Dessa forma, a lista encadeada nos permite lidar com a flexibilidade do tamanho do caminho, alocando memória apenas conforme necessário.
- Uma vantagem da utilização da lista encadeada para a nossa lista de passos é que não há necessidade de percorrer, buscar por índices ou ordenar o array. Essas operações são custosas em termos de desempenho quando lidamos com uma lista encadeada.

```c
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include <...>

typedef struct linkedListNode {
  Position position;
  Node *node;
  struct linkedListNode *next;
  struct linkedListNode *prev;
} LinkedListNode;

typedef struct linkedList {
  struct linkedListNode *head;
  struct linkedListNode *tail;
  int size;
} LinkedList;

LinkedList *LinkedListCreate();
LinkedListNode *LinkedListNodeCreate(Position position);
LinkedListNode *LinkedListNodeCreateByNode(Node *);

void LinkedListClear(LinkedList *list);
void LinkedListDestroy(LinkedList **list);

bool LinkedListInsertEnd(LinkedList *, Position, Node *);
bool LinkedListInsertStart(LinkedList *, Position, Node *);

bool LinkedListRemoveEnd(LinkedList *, Position *, Node **);
bool LinkedListRemoveStart(LinkedList *, Position *, Node **);

bool LinkedListIsEmpty(LinkedList *list);
bool LinkedListHasValue(LinkedList *list, Position position);

#endif  // LINKED_LIST_H
```

Código 2: Implementação do TAD de lista (lista ligada).

2.1.3 Maze (ou Labirinto)

O TAD do labirinto foi utilizado de forma dinâmica no projeto, representando o tabuleiro do labirinto utilizando os caracteres "#" ou "*" para representar paredes e o caractere " "(espaço em branco) para indicar possíveis caminhos. Foram associadas ao labirinto funções como a obtenção de possíveis caminhos a partir de uma posição específica e a localização da posição do rato para determinar sua posição inicial.

```
#ifndef MAZE_H
#define MAZE_H

#include<...>

typedef struct maze {
  char** board;
  int qtyLines;
  int qtyColumns;
  char option;
} Maze;

Maze* MazeCreate();
void MazeRead(Maze* maze);

Position MazeGetMousePosition(Maze* maze);
Queue* MazeGetAvailablePositions(Maze* maze, Position position, Path* path);
bool MazeIsPositionExit(Maze* maze, Position pos);

void MazePrintWithMarkers(Maze* maze, LinkedList* markers);
void MazeDestroy(Maze** maze);

#endif  // MAZE_H
```

Código 3: Implementação do TAD do labirinto.

2.1.4 Path (ou Caminho)

A implementação da TAD de caminho, foi dada como a adição de funções específicas a lista ligada.

```
#ifndef PATH_H
#define PATH_H

#include <...>

typedef LinkedList Path;

Path* PathCreate();

bool PathAddStep(Path* path, Position position);
bool PathRemoveStep(Path* path, Position* position);
bool PathPercourNext(Path* path, Position* position);
bool PathIsEmpty(Path* path);

void PathCopy(Path* dest, Path* origin);
void PathDestroy(Path** path);

#endif  // PATH_H
```

Código 4: Implementação do TAD de caminho.

2.1.5 Queue (ou Fila)

A implementação da Fila, foi dada na adição de funções específicas a lista ligada.

```c
#ifndef QUEUE_H
#define QUEUE_H

#include <...>

typedef LinkedList Queue;

Queue *QueueCreate();
bool QueuePush(Queue *, Position);
bool QueuePushByNode(Queue *, Node *);
bool QueuePop(Queue *, Position *);
bool QueuePopByNode(Queue *, Node **);
bool QueueIsEmpty(Queue *);
void QueueDestroy(Queue **stack);

#endif  // QUEUE_H
```

Código 5: Implementação do TAD de fila.

2.1.6 Tree (ou Árvore)

Como foco do trabalho, foi implementada a TAD árvore, a qual usamos para armazenar a árvore mapear o labirinto e encontrar a saida.

```c
#ifndef QUEUE_H
#define QUEUE_H

#include <...>

typedef struct {
  Node* root;
  Position mazeExit;
} Tree;

Tree* TreeCreate();

void TreePrintNodesByLevel(Tree* tree);
Tree* MapMazeRecursivly(Maze* maze);
void PrintNodesByLevel(Node* root);
bool TreeMazeFindExit(Tree* tree, Path* shortestPath, Path* biggestPath);

#endif  // QUEUE_H
```

Código 6: Implementação do TAD de árvore.

2.2 Funções principais

Após a implementação das TADs e funções auxiliares, procedemos à compreensão e implementação das funções principais, que são responsáveis por encontrar a saída do labirinto.

2.2.1 Mapeamento do labirinto em uma árvore

A função de de mapeamento do labirinto em uma arvore, usamos a implementação do algoritmo de busca em profundidade, e assim pra cada nó visitamos os disponíveis e pra cada um que visitamos salvamos-os na arvore.

```
Tree* MapMazeRecursivly(Maze* maze) {
  Path* currentPath = PathCreate();
  Tree* tree = TreeCreate();
  tree->mazeExit = PositionCreate(maze->qtyColumns - 1, maze->qtyLines - 2);
  Position mousePosition = MazeGetMousePosition(maze);
  MapMaze(maze, mousePosition, currentPath, &(tree->root), NULL);
  PathDestroy(&currentPath);
  return tree;
}

void MapMaze(Maze* maze, Position pos, Path* currentPath, Node** currentNode,
             Node* fatherNode) {
  *currentNode = NodeCreate(ItemCreate(pos), fatherNode);
  PathAddStep(currentPath, pos);

  Queue* availablePositions = MazeGetAvailablePositions(maze, pos, currentPath
      );
  Position availablePosition;
  while (QueuePop(availablePositions, &availablePosition)) {
    Node** auxNode =
        NodeGetNodeByPositionDirection(*currentNode, availablePosition);
    MapMaze(maze, availablePosition, currentPath, auxNode, *currentNode);
    PathRemoveStep(currentPath, &availablePosition);
  }

  LinkedListDestroy(&availablePositions);
}
```

Código 7: Implementação das funções de mapeamento do labirinto em uma árvore.

2.2.2 Encontrar saida do labirinto usando a árvore

Para encontrar a saida usamos ainda a implementação da busca em profundidade, implícita pela recursão.

```
void TreeMazeFindExitRecursivly(Position exitMazePosition, Node* node,
                                Path* shortestPath, Path* biggestPath,
                                Path* currentPath) {
  PathAddStep(currentPath, node->item.position);
  if (exitMazePosition.x == node->item.position.x &&
      exitMazePosition.y == node->item.position.y) {
    if (currentPath->size < shortestPath->size)
      PathCopy(shortestPath, currentPath);
    if (currentPath->size > biggestPath->size)
      PathCopy(biggestPath, currentPath);
  }
  if (NodeIsLeaf(node)) return;
  Position p;
  if (node->r != NULL) {
    TreeMazeFindExitRecursivly(exitMazePosition, node->r, shortestPath,
                               biggestPath, currentPath);
    PathRemoveStep(currentPath, &p);
  }
  if (node->d != NULL) {
    TreeMazeFindExitRecursivly(exitMazePosition, node->d, shortestPath,
                               biggestPath, currentPath);
    PathRemoveStep(currentPath, &p);
  }
  if (node->l != NULL) {
    TreeMazeFindExitRecursivly(exitMazePosition, node->l, shortestPath,
                               biggestPath, currentPath);
    PathRemoveStep(currentPath, &p);
  }
  if (node->u != NULL) {
    TreeMazeFindExitRecursivly(exitMazePosition, node->u, shortestPath,
                               biggestPath, currentPath);
    PathRemoveStep(currentPath, &p);
  }
}

bool TreeMazeFindExit(Tree* tree, Path* shortestPath, Path* biggestPath) {
  shortestPath->size = INT_MAX;
  biggestPath->size = INT_MIN;

  Path* currentPath = PathCreate();

  TreeMazeFindExitRecursivly(tree->mazeExit, tree->root, shortestPath,
      biggestPath, currentPath);

  shortestPath->size--;
  biggestPath->size--;
  PathDestroy(&currentPath);
  return (shortestPath->size != INT_MAX - 1);
}
```

Código 8: Implementação das funções de achar saida pela arvore.

Foram criadas funções que é chamada pelo usuário, e funções recursiva separada, a fim de evitar que o usuário precise fornecer informações além do necessário, como o caminho atual, que não faz sentido fora da recursão. Essa abordagem permite uma interação mais simplificada por parte do usuário, enquanto a função recursiva lida com a lógica de busca e percurso no labirinto.

3 Estudo de complexidade

Como estamos em um labirinto, onde podemos andar nas direções cima, baixo, esquerda ou direita, e em um pior caso onde andamos para uma dada posição, aparece mais três possíveis posições (exceto no primeiro movimento ele tem 4 possíveis caminhos), isto considerando que ele não pode revisitar as casas que ele ja visitou.
Assim chegamos que a ordem de complexidade dos algoritmos é de $O(3^N)$, sendo N a quantidade de linhas x quantidade de colunas, isto para ambos os algoritmos dado que o funcionamento é o mesmo, só muda o foco de busca de cada um.

4 Testes

Submetemos o programa a alguns testes, e conseguimos obter os resultados esperados. Extraindo um dos casos de testes, conseguimos ver os comportamentos propostos.

```
11 11
***********
*M*       *
*   *** * *
*   *     *
*** * * * *
* *   *   *
* * * *   *
*       * *
* *** * * *
*       *
***********
```

Código 9: Tabuleiro submetido.

```
47
***********
*M*.......*
*...*** *.*
* ..*... .*
*** *.*.*.*
* *  .*...*
* * *.*...*
*.....  .*.*
*.*** *.*.*
*.......*..
***********
```

Código 10: Maior caminho encontrado.

```
19
***********
*M*       *
*...*** * *
*  .*     *
***.* * * *
* *...*   *
* * *.*...*
*    ...*.*
* *** * *.*
*       *..
***********
```

Código 11: Menor caminho encontrado.

Como esperado, foi possível obter o menor e o maior caminho, utilizando o mapeamento do labirinto na árvore.

5 Análise

Ao compararmos a estratégia, utilizada nesse trabalho possuí vantagens e desvantagens em relação ao trabalho anterior, este trabalho possuí uma complexidade um pouco maior, porém, conseguimos mapear todos os caminhos, ainda obter dados mais detalhados, como quantos caminhos possuí até a saida, menor e maior caminho, caminhos que levam a paredes entre outros.

5.1 Comparação dos algoritmos

A implementação pela árvore apesar de possuir muitas vantagens, ela é mais custosa em relação a conhecimento, e em relação a custo de memória, tendo em vista que pra mapear o labirinto, é necessário armazenar todas as posições do labirinto na arvore.

6 Conclusão

Com o desenvolvimento do trabalho, conseguimos entender exatamente o funcionamento do primeiro programa implementado na primeira parte deste trabalho, ainda conseguimos fixar o conhecimento das estruturas de dados pilhas, filas e árvores, com uma aplicação prática dos mesmos.

7 Referências

Aulas teóricas e práticas de Estrutura de Dados I.