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
9 de julho de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

No presente relatório, serão detalhados os passos envolvidos no desenvolvimento do segundo 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 na resolução de um labirinto, onde se buscava encontrar a saída a partir de uma posição inicial, mediante a disponibilização de um labirinto pré-determinado.

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.

Para a implementação dos conceitos de busca em largura e busca em profundidade utilizamos os conceitos das estruturas:

- Fila estrutura onde o primeiro a entrar é o primeiro a sair (First in, first out)
- Pilha estrutura onde o ultimo a entrar é o primeiro a sair (Last in, first out)

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.
- *Allegro5*: Biblioteca para implementação da interface gráfica.

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 Dependências

Neste projeto utilizamos a biblioteca allegro 5, e como o projeto foi desenvolvido no ambiente linux é necessário instala-lo:

Instalação das dependências

```
sudo add-apt-repository ppa:allegro/5.2
sudo apt update
sudo apt-get install liballegro*5.2 liballegro*5-dev
```

Caso não esteja no ambiente linux confira a documentação do allegro clicando aqui.

1.6 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 – sem a interfaçe grafica
./exe ui < entrada.in – com a interface grafica
```

Ainda foi desenvolvido um shell script para a execução dos testes dentro da pasta test-cases, o script nomeado ./runtests.sh na raiz do projeto.

Executando os testes automatizados

```
chmod +x ./runtests.sh
./runtests.sh
```

2 Implementação

Iniciamos o processo de desenvolvimento do trabalho criando as TADs (Tipos Abstratos de Dados) para as listas e, em seguida, implementamos as demais TADs necessárias, juntamente com suas respectivas funções. Posteriormente, prosseguimos com a implementação das três principais funções que desempenham um papel fundamental na busca da saída do labirinto.

2.1 Tipos abstratos de dados (TADS)

Implementamos seguintes TADS:

- Position (Responsável por armazenar e manipular uma cordenada (X,Y)).
- Maze (Responsável por armazenar e manipular os dados de um labirinto, como quantidade de linhas, colunas e o tabuleiro)
- LinkedList & LinkedListNode (Implementação de uma lista ligada, a LinkedListNode é a celula da lista ligada)
- Stack (Implementação da pilha (LIFO))
- Queue (Implementação da fila (FIFO))

2.1.1 Position (ou Posição)

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

```c
#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);

#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;
  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);

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

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

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

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 Stack (ou Pilha)

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

```c
#ifndef STACK_H
#define STACK_H

#include <...>

typedef LinkedList Stack;

Stack *StackCreate();

bool StackPush(Stack *, Position);
bool StackPop(Stack *, Position *);
bool StackIsEmpty(Stack *stack);
void StackDestroy(Stack **stack);

#endif  // STACK_H
```

Código 5: Implementação do TAD de pilha.

2.1.6 Queue (ou Fila)

A implementação da Fila, como a stack e a pilha, 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 QueuePop(Queue *, Position *);
bool QueueIsEmpty(Queue *);
void QueueDestroy(Queue **stack);

#endif  // QUEUE_H
```

Código 6: Implementação do TAD de path.

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 Busca em largura e busca em profundidade

Tanto o algoritmo de busca em largura quanto o de busca em profundidade possuem um pseudo código semelhante, diferenciando-se apenas na forma de armazenar os elementos. Na busca em largura, os elementos são armazenados em uma fila, enquanto na busca em profundidade, eles são armazenados em uma pilha.

1. Empilha/Enfilera a posição inicial
2. Desempilha/Desenfilera a posição
3. Se a posição for nula, retorna nulo
4. Se a posição desempilhado/desenfileirada não for nula, empilha as posições disponiveis a partir da posição desempilhada/desenfileirada

E repete-se os passos 2, 3 e 4.

2.2.2 Busca em Profundidade (por pilha)

A busca em profundidade é um algoritmo que percorre um caminho por vez. Ele entra em um caminho e, seguindo a ordem de empilhamento, percorre esse caminho até alcançar o seu final antes de voltar e explorar outros caminhos. Devido a essa característica de percorrer caminhos o mais profundamente possível antes de retornar, o algoritmo é chamado de busca em profundidade. A seguir, apresentamos a implementação desse algoritmo:

```
bool FindPathStack(Maze* maze, Path* path) {
  // We initilized-it with -1 to disconsider the initial mouse movement.
  path->size--;

  Position mousePosition = MazeGetMousePosition(maze);
  Stack* stack = StackCreate();

  StackPush(stack, mousePosition);

  while (!StackIsEmpty(stack)) {
    Position pos;
    bool itemRemoved = StackPop(stack, &pos);

    if (!itemRemoved) break;
    PathAddStep(path, pos);

    if (MazeIsPositionExit(maze, pos)) {
      StackDestroy(&stack);
      return true;
    }

    Queue* availablePositions = MazeGetAvailablePositions(maze, pos, path);
    Position positionAvailable;
    while (QueuePop(availablePositions, &positionAvailable))
      StackPush(stack, positionAvailable);

    LinkedListDestroy(&availablePositions);
  }

  StackDestroy(&stack);

  return false;
}
```

Código 7: Implementação da busca em profundidade por pilha.

2.2.3 Busca em Largura (por fila explicita)

A busca em largura, ao contrário da busca em profundidade, percorre vários caminhos simultaneamente. Podemos imaginar que ela se ramifica nas bifurcações do labirinto, daí o nome "busca em largura", pois o objetivo é explorar ao longo de todo o labirinto ao mesmo tempo, garantindo o mapeamento completo do mesmo.
Com este é importante observar que ao achar uma solução, ela será encontrada junto ao menor caminho, pois este percorre todos os caminhos em paralelo, ou seja, ele percorre todas as linhas em vez de aprofundar em um caminho especifico, assim conseguindo garantir o menor caminho.

```
bool FindPathQueue(Maze* maze, Path* path) {
  // We initilized-it with -1 to disconsider the initial movement of mouse.
  path->size--;

  Position mousePosition = MazeGetMousePosition(maze);
  Queue* queue = QueueCreate();
  QueuePush(queue, mousePosition);

  while (!QueueIsEmpty(queue)) {
    Position pos;
    bool itemRemoved = QueuePop(queue, &pos);

    if (!itemRemoved) break;
    PathAddStep(path, pos);

    if (MazeIsPositionExit(maze, pos)) {
      QueueDestroy(&queue);
      return true;
    }

    Queue* availablePositions = MazeGetAvailablePositions(maze, pos, path);
    Position availablePosition;
    while (QueuePop(availablePositions, &availablePosition))
      QueuePush(queue, availablePosition);

    LinkedListDestroy(&availablePositions);
  }

  QueueDestroy(&queue);

  return false;
}
```

Código 8: Implementação da busca em largura por fila.

2.2.4 Busca em Profundidade (por pilha implicita buscando o menor caminho)

Além disso, importamos o código do primeiro trabalho e fizemos as devidas alterações para implementar a busca em profundidade, utilizando a abordagem implícita através da pilha de chamadas. Dessa forma, adaptamos e incorporamos o código anterior para utilizar a estratégia da busca em profundidade em nosso novo projeto.
Nesta implementação é importante observar que, apesar de ser uma busca em profundidade, percorremos ainda todo o labirinto, procurando pelo menor caminho. Ou seja, a pilha foi implicitamente implementada pela pilha de chamadas, já a fila foi implementada pela fila de caminhos possíveis.

```
void FindPath(Maze* maze, Position pos, Stack* currentPath, Path* pathFound,
              Path* pathPercourred);

bool FindPathRecursivly(Maze* maze, Path* pathFound, Path* pathPercourred) {
  Stack* currentPath = PathCreate();
  LinkedListClear(pathFound);
  pathFound->size = INT_MAX;

  Position mousePosition = MazeGetMousePosition(maze);
  FindPath(maze, mousePosition, currentPath, pathFound, pathPercourred);

  bool foundPath = !LinkedListIsEmpty(pathFound);
  if (!foundPath) PathCopy(pathFound, currentPath);

  // We initilized-it with -1 to disconsider the initial movement of mouse.
  currentPath->size--;
  pathFound->size--;
  pathPercourred->size--;

  StackDestroy(&currentPath);

  return foundPath;
}

void FindPath(Maze* maze, Position pos, Stack* currentPath, Path* pathFound,
              Path* pathPercourred) {
  StackPush(currentPath, pos);
  PathAddStep(pathPercourred, pos);
  if (MazeIsPositionExit(maze, pos) && currentPath->size < pathFound->size)
    return PathCopy(pathFound, currentPath);

  Queue* availablePositions = MazeGetAvailablePositions(maze, pos, currentPath
      );
  Position availablePosition;
  while (QueuePop(availablePositions, &availablePosition)) {
    FindPath(maze, availablePosition, currentPath, pathFound, pathPercourred);
    StackPop(currentPath, &availablePosition);
  }

  LinkedListDestroy(&availablePositions);
}
```

Código 9: Implementação da busca em profundidade pela pilha de chamadas.

Foi criada uma função que é chamada pelo usuário, e outra função recursiva separada, a fim de evitar que o usuário precise fornecer informações além do necessário, como a posição inicial do rato. 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.

```
**********
*          *
******** *
*M*   *   *
* *** * * *
*
**********
```

Código 10: Tabuleiro submetido.

```
11
**********
*          *
******** *
*M*   *   *
*o*** * * *
*oooooooooo
**********
```

Código 11: Saida da implementação por recursão.

```
17
**********
*          *
******** *
*M*ooo*o  *
*o***o*o* *
*oooooooooo
**********
```

Código 12: Saida da implementação por fila.

```
30
**********
*oooooooo*
*********o*
*M*ooo*ooo*
*o***o*o*o*
*oooooooooo
**********
```

Código 13: Saida da implementação por pilha.

Conforme observado anteriormente, com a lista implementada por recursão, conseguimos obter o menor caminho possível (que foi implementado na primeira parte do trabalho e adaptado para o novo código). No entanto, em nossa implementação, a lista é interrompida assim que encontramos a saída, o que significa que o mapeamento completo do labirinto não é realizado. Por outro lado, ao utilizar a pilha, é possível realizar o mapeamento completo do labirinto.

5 Análise

Ao compararmos os algoritmos implementados, podemos observar que cada um possui suas vantagens e desvantagens em relação às respectivas implementações.

5.1 Comparação dos algoritmos

As implementações utilizando fila e recursão são mais custosas, pois elas mapeiam todo o labirinto. No entanto, essas abordagens nos permitem obter exatamente o menor caminho, sem interferência de outros caminhos

Na implementação utilizando pilha, não conseguimos garantir o menor caminho devido ao fato de o algoritmo explorar o caminho mais profundo antes de retornar. Isso impede que o mapeamento completo do labirinto seja obtido. No entanto, essa abordagem é mais otimizada, pois não requer uma varredura completa de todo o labirinto.

Além disso, é importante levar em consideração as chamadas na pilha de execução (call stack), principalmente quando lidamos com problemas de grande escala. A estratégia de passagem por valor, utilizada na implementação recursiva, pode resultar em um consumo significativo de memória, especialmente se o problema for complexo e exigir muitas chamadas recursivas.

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 lista, pilhas e filas, com uma aplicação prática dos mesmos.

7 Referências

Aulas teóricas e práticas de Estrutura de Dados I.