Algoritmos e Estruturas de Dados LC 2016/2

Exercício Programa 2: Labirinto

Rodrigo de Souza

Entrega: 14/11/2016

1 Enunciado

Nosso objetivo neste Exercício-Programa é implementar um método para ajudar Joãozinho e Maria ou quem quer que seja que esteja perdido em um labirinto e queira encontrar a saída.

A entrada do problema é uma matriz quadrada $n \times n$ representando o labirinto. Cada posição vale 0 ou 1, onde 1 indica a presença de uma parede, que não pode ser atravessada nem ocupada pelo indivíduo. Ou seja, uma posição parede não pode ser usada em um possível trajeto para sair do labirinto, as posições que valem 0 são livres e podem ser usadas.

Além da matriz, faz parte da entrada dois pares de inteiros representando posições: o primeiro par (x, y) indica a posição inicial do indivíduo perdido, linha x e coluna y da matriz e o segundo par, (x', y'), indica a posição de uma porta de saída (vamos assumir que nenhuma das duas é uma parede).[1]

Seu algoritmo deve imprimir um roteiro que leve o indivíduo à saída: uma sequência de posições (x_1, y_1), (x_2, y_2), $\ldots$, (x_k, y_k), sem passar por nenhuma parede, onde (x_1, y_1) é a posição inicial, ou seja, (x, y), (x_k, y_k) é a posição final, ou seja, (x', y'), e duas posições contíguas (x_j, y_j), (x_{j+1}, y_{j+1}) precisam ser adjacentes na matriz: (x_{j+1}, y_{j+1}) deve estar à direita, à esquerda, acima ou abaixo de (x_j, y_j) (não é permitido andar na diagonal).

Sua solução consiste, de forma intuitiva, em sair andando no labirinto, selecionando a cada passo uma posição viável – livre de parede, e *ainda não visitada*, e marcar essa posição, para que você saiba que já passou por ela (como Joãozinho e Maria fizeram jogando migalhas de pão no caminho). Possivelmente, você vai chegar em um beco sem saída – todas as posições adjacentes são ou paredes ou posições já visitadas. Neste caso, sua solução executa o que se chama *backtracking*: ela volta um passo no caminho que construiu até agora, e tenta outra direção, a partir do passo anterior (e caso não haja outra direção no passo anterior, faz um novo backtracking, etc.) Como fazemos para armazenar a sequência de posições anteriores, para que o algoritmo possa executar o backtracking? Aqui entra uma estrutura de dados importante: essas posições devem ser armazenadas em uma pilha.

[1] Você pode escolher se os índices começam em 0 ou 1, mas isso deve ficar bem claro no seu programa.

De maneira mais formal, seu algoritmo mantem uma pilha de posições. Ele começa empilhando a posição inicial (x, y). Em cada iteração, seu algoritmo considera a posição no topo da pilha, e procura a próxima posição adjacente ainda não visitada. Para ser feito com critério, vamos supor que seu algoritmo considera as posições sempre na mesma sequência: *acima*, *direita*, *abaixo*, *esquerda*. Seu algoritmo procura assim a próxima posição viável (não foi ainda visitada, nem é uma parede) nessa sequência; caso encontre, anda nessa direção, ou seja: empilha a nova posição visitada, e marca essa posição na matriz (pode marcar com qualquer símbolo diferente de 0 e 1 – marcar é o mesmo que jogar uma migalha de pão para você saber que já andou por lá). Caso seja um beco sem saída (não encontrou nenhuma posição viável), seu algoritmo desempilha a posição no topo, e tenta continuar a partir da posição anterior (que é o novo topo da pilha).

Ao encontrar a saída, seu programa imprime o trajeto. Como ele faz isso usando a pilha de posições? Caso não haja trajeto possível, seu programa imprime a mensagem "sem saída".

2 Instruções gerais

- Seu programa deve ser feito em C.
- Seu programa deve ler o labirinto de um arquivo `labirinto.dat` com o seguinte formato: a primeira linha é a dimensão da matriz (um número inteiro), ou seja, o número de linhas e colunas do labirinto; a segunda linha tem dois inteiros, separados por espaço, representando a posição inicial; a terceira linha, dois inteiros, separados por espaço, representando a posição de saída; as próximas n linhas codificam a matriz: cada uma tem uma sequência de n dígitos 0's e 1's, separados por espaço.
- Não tente embelezar a saída do seu programa com mensagens, formatações, etc. Seu programa será julgado segundo a adequação de sua saída à descrição do exercício.
- Você deve implementar um módulo de tratamento de pilha da forma que estudamos na aula: você deve entregar, além de um arquivo com o programa principal, mais dois arquivos, `pilha.h` com a definição da estrutura pilha e os cabeçalhos de funções, e um arquivo `pilha.c` com a implementação das funções de manipulação de pilha. Note que você também deverá implementar um tipo de dados para posições da matriz (tipicamente usando `struct`). Sua pilha usará esse tipo de dados (será uma pilha de posições, cabe recordar). A leitura dos dados e o algoritmo de busca estão no arquivo principal. Nesse arquivo principal, você deve criar uma pilha e usar as funções definidas em `pilha.c` para manipulá-la.
- Documente cada função dizendo o quê ela faz. Se o seu código não está legível, fica muito difícil adivinhar o que a função faz e isso prejudica a correção. Esse comentário deve especificar o que a função recebe (os parâmetros) e o que devolve.
- Capriche. Cuidado com a indentação. Deixei seu programa suficientemente modularizado para que cada tarefa específica esteja implementada

em uma função (documentada explicando o que recebe e o que faz). Faça isso em particular para as estruturas de dados, implementando funções que realizam as operações pertinentes. Modularização e capricho serão considerados na nota.

- Escreva no início do código um cabeçalho com comentários, indicando nome, número do EP, data, nome da disciplina.
- A entrega será eletrônica no ambiente Moodle da disciplina (não receberei exercícios impressos ou via email).