CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Aplicação de Pilha
Parte 3

Disciplina: Estrutura de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Aplicações de Pilhas (Stacks)

- Estudaremos uma aplicação de pilhas:
  - *Backtracking*.
    - Problema de percurso de um labirinto (Maze).

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto

- Um problema clássico que requer o uso de uma pilha é o problema de encontrar um caminho em um labirinto (Maze Problem).

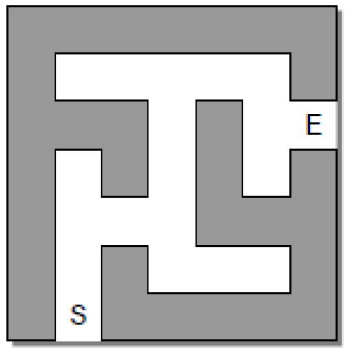


Figure 7.5: A sample maze with the indicated starting (S) and exit (E) positions.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto

- O algoritmo deve explorar o labirinto até achar a posição de saída conforme ilustra a figura.

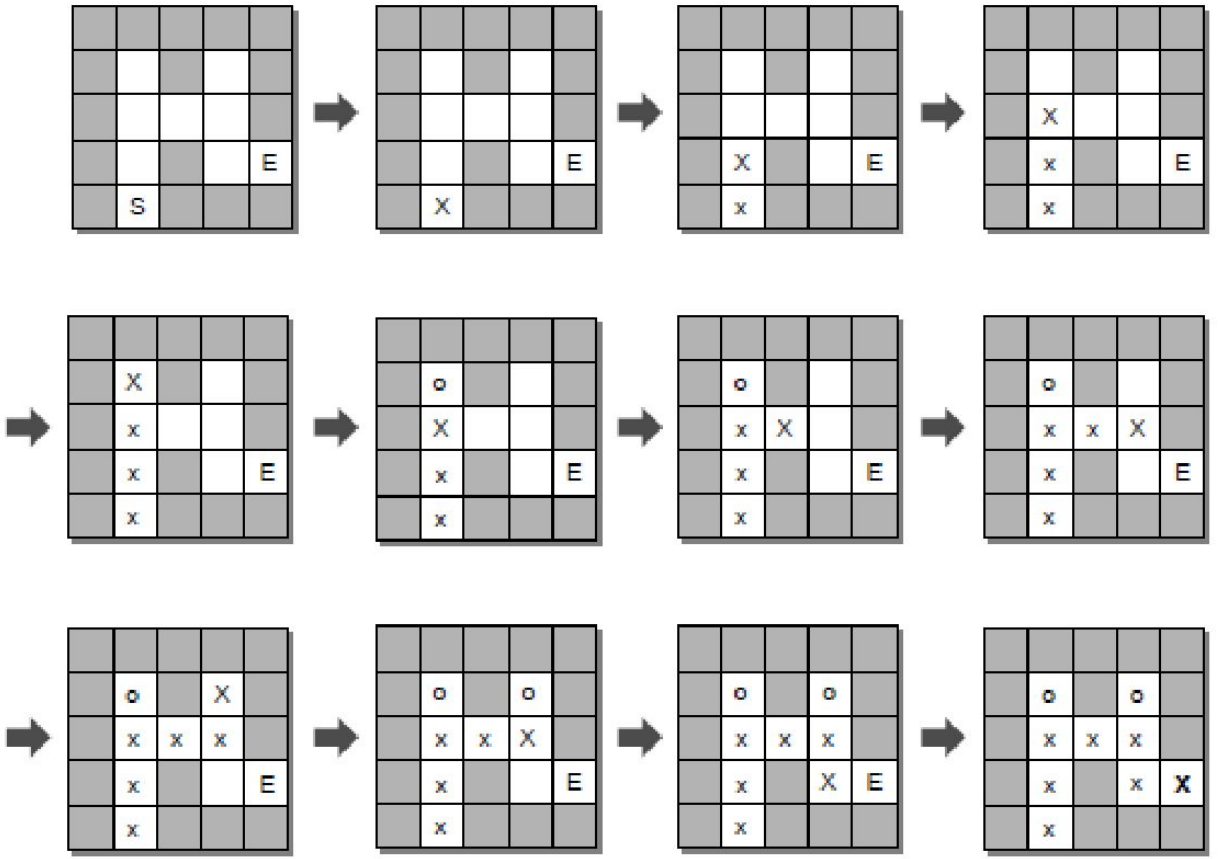


Figure 7.9: The sequence of steps taken through a sample maze. `S` is the starting position and `E` is the exit. The path is marked using the character `x` while the current cell is marked with `X`. Cells we visited but from which we had to backtrack are marked with `o`.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto

- O algoritmo para resolver este problema segue a estratégia de ***backtracking***.
- Ao invés de tentar todos os caminhos possíveis, o que corresponderia ao método de força bruta, o algoritmo armazena as soluções parciais de caminhos possíveis, e sempre que o caminho se torna inviável por causa de uma passagem bloqueada, tenta aproveitar parte desse caminho seguindo por uma nova direção. Para isso, se faz uso de uma pilha.
- Assim, o algoritmo evita construir uma solução nova desde o inicio.
- O algoritmo constrói soluções parciais um passo de cada vez.
- Pode-se considerar que o método de *backtracking* é um refinamento do método de força bruta.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- Define-se um TAD Labirinto (Maze ADT) para construir e resolver um labirinto. O labirinto é uma estrutura bidimensional formada por células de igual tamanho distribuída em linhas e colunas.
  - Cada célula pode ser preenchida representar uma parede ou um espaço vazio;
  - Uma célula deve ser marcada para representar a posição de partida e outra para representar a saída.

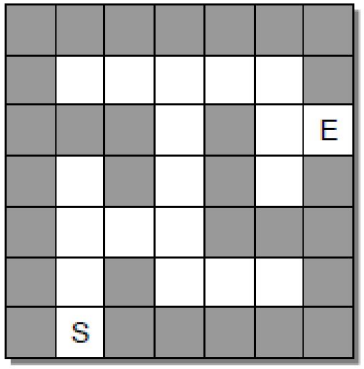


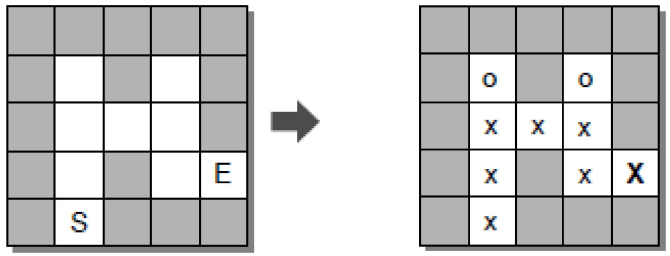
Figure 7.6: Sample maze from Figure 7.5 divided into equal-sized cells.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) possui as seguintes operações:
  - **Maze (numRows, numCols) :** Cria um novo labirinto com todas as células vazias. Sem ponto de partida ou saída.
  - **numRows(row, col) :** Retorna o número de linhas do labirinto.
  - **numCols(row, col) :** Retorna o número de colunas do labirinto.
  - **setWall(row, col) :** Define a célula (row, col) como parede.
  - **setStart(row, col) :** Define a célula (row, col) como ponto de partida.
  - **setExit(row, col) :** Define a célula (row, col) como a saída.
  - **findPath() :** Tenta resolver o labirinto encontrando um caminho do ponto de partida até a saída. Caso a solução seja encontrada, o caminho é marcado com tokens (x) e retorna True. Caso contrário, o labirinto é mostrado no estado inicial e retorna False. O perímetro do labirinto pode ser vazio, entendendo-se que existe uma fronteira além dele.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) possui as seguintes operações:
  - **reset() :** Apaga os tokens (x) colocados durante a operação de busca.
  - **draw() :** Imprime uma representação do labirinto usando caracteres. Devem ser mostrados: paredes, ponto de partida e saída e caminho.



- Nesta versão, a solução do labirinto consiste em simplesmente marcar o caminho desde o ponto de partida até a saída. Outras versões de propósito geral podem retornar o caminho solução como uma sequência de coordenadas a serem seguidas.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – solvemaze

- Usando o TAD Labirinto (Maze ADT) podemos construir um programa para construir e resolver um labirinto.
- A lógica do programa principal é simples:
  - Construir os labirinto;
  - Determinar se o caminho de saída existe;
  - Imprimir o labirinto se o caminho existe.

Listing 7.4 The `solvemaze.py` program.

```
1  # Program for building and solving a maze.
2  from maze import Maze
3
4  # The main routine.
5  def main():
6    maze = buildMaze( "mazefile.txt" )
7    if maze.findPath() :
8      print( "Path found...." )
9      maze.draw()
10   else :
11     print( "Path not found...." )
12
```

Construir labirinto

Encontrar caminho

Desenhar labirinto

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – solvemaze

- A função buildMaze() constrói o labirinto a partir de um arquivo de texto, com o seguinte formato:

```
5 5
4 1
3 4
*****
*.*.*
*...*
*.*..
*.***
```

- A primeira linha contem as dimensões do labirinto;
- As duas linhas seguintes contem as coordenadas do ponto de partida e da saída.
- As linhas seguintes representam o labirinto com * representando as paredes.

```
13  # Builds a maze based on a text format in the file.
14  def buildMaze( filename ):
15    infile = open( filename, "r" )
16
17      # Read the size of the maze.
18    nrows, ncols = readValuePair( infile )
19    maze = Maze( nrows, ncols )
20
21      # Read the starting and exit positions.
22    row, col = readValuePair( infile )
23    maze.setStart( row, col )
24    row, col = readValuePair( infile )
25    maze.setExit( row, col )
26
27      # Read the maze itself.
28    for row in range( nrows ) :
29      line = infile.readline()
30      for col in range( len(line) ) :
31        if line[col] == "*" :
32          maze.setWall( row, col )
33
34     # Close the maze file and return the newly maze.
35    infile.close()
36    return maze
```

Leitura de 2 valores

Abre o arquivo

Constrói o labirinto

Define o Inicio

Define a Saída

Define uma célula como parede

Fecha o arquivo

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – solvemaze

- A função readValuePair() faz a leitura de dois valores de uma linha do arquivo de texto, e retorna como valores inteiros.

```
37
38  # Extracts an integer value pair from the given
39  def readValuePair( infile ):
40    line = infile.readline()
41    valA, valB = line.split()
42    return int(valA), int(valB)
43
44  # Call the main routine to execute the program.
45  main()
```

Divisão da linha em dois

Leitura de uma linha

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) é implementado com um vetor bidimensional, importando a classe Array2D.
- Caracteres podem ser usados para representar as paredes e os tokens (x) de caminho válido e (o) de tentativa falha.
- Já células vazias são representadas como ponteiros nulos.
- Enquanto as posições de partida e saída são armazenadas separadamente.

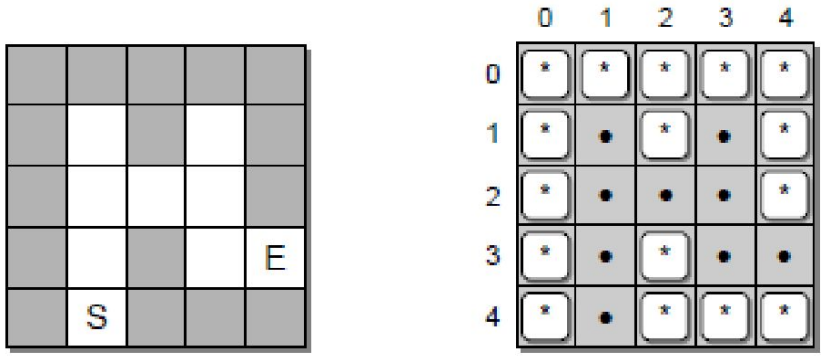


Figure 7.11: The abstract view of a maze physically represented using a 2-D array. Walls are indicated with an asterisk (*) character, while open cells contain a null reference. The start and exit cells will be identified by cell position stored in separate data fields.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) é implementado com um vetor bidimensional, importando a classe Array2D.
- Além disso, ele utiliza a implementação a classe Stack.

Listing 7.5 The maze.py module.

```
1  # Implements the Maze ADT using a 2-D array.
2  from array import Array2D
3  from liststack import Stack
4
5  class Maze :
6     # Define constants to represent contents of the maze
7    MAZE_WALL = "*"
8    PATH_TOKEN = "x"
9    TRIED_TOKEN = "o"
10
11    # Creates a maze object with all cells marked as open
12   def __init__( self, numRows, numCols ):
13     self._mazeCells = Array2D( numRows, numCols )
14     self._startCell = None
15     self._exitCell = None
16
17    # Returns the number of rows in the maze.
18   def numRows( self ):
19     return self._mazeCells.numRows()
20
21    # Returns the number of columns in the maze.
22   def numCols( self ):
23     return self._mazeCells.numCols()
24
```

Constructor

Número de linhas

Número de colunas

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) possui métodos para definir as células que serão: i) paredes; ii) ponto de partida; e iii) ponto de saída.

Define as paredes

Define o ponto de partida

Define o ponto de saída

```
25     # Fills the indicated cell with a "wall" marker.
26    def setWall( self, row, col ):
27      assert row >= 0 and row < self.numRows() and \
28             col >= 0 and col < self.numCols(), "Cell index out of range."
29      self._mazeCells.set( row, col, self.MAZE_WALL )
30
31     # Sets the starting cell position.
32    def setStart( self, row, col ):
33      assert row >= 0 and row < self.numRows() and \
34             col >= 0 and col < self.numCols(), "Cell index out of range."
35      self._startCell = _CellPosition( row, col )
36
37     # Sets the exit cell position.
38    def setExit( self, row, col ):
39      assert row >= 0 and row < self.numRows() and \
40             col >= 0 and col < self.numCols(), \
41             "Cell index out of range."
42      self._exitCell = _CellPosition( row, col )
```

Preenche uma posição da matriz com um *

Repassa uma coordenada

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) possui métodos para:
  - i) encontrar o caminho entre o ponto de partida e a saída;
  - ii) remover todos os tokens (x) na matriz do labirinto;
  - iii) imprimir uma representação do labirinto usando caracteres.

Procura o caminho no labirinto

Apaga os tokens (x)

Imprime o labirinto

```
43
44      # Attempts to solve the maze by finding a path from the starting cell
45      # to the exit. Returns True if a path is found and False otherwise.
46      def findPath( self ):
47        ......
48
49      # Resets the maze by removing all "path" and "tried" tokens.
50      def reset( self ):
51        ......
52
53      # Prints a text-based representation of the maze.
54      def draw( self ):
55        ......
```

Aplicações de Pilhas (Stacks)
Função findpath()

- A função **findpath()** tenta resolver o labirinto encontrando um caminho do ponto de partida até a saída. Nesse caso retorna True. Caso contrário, False.
- A função utiliza uma pilha para guardar as posições que conformam o caminho. E se necessário fazer ***backtrack***.

```
def findPath(self):
    s = Stack()                    # Cria a Pilha
    s.push(self._startCell)        # Guarda na pilha
    self._markPath(self._startCell.row, self._startCell.col) # Marca caminho
    while True:
        #inside the loop where neighbor cell is checked for valid move
        #same time, checked if we get the exit
        if s.estaVazia():          # Voltamos até o ponto de partida
            return False           # Não existe caminho
        current_cell = s.peek() # Define a celula atual com o topo da pilha
        row = current_cell.row
        col = current_cell.col
```

Cria a pilha e guarda a posição de partida

Inicia um loop de busca da próxima posição até achar a saída o voltar a posição de partida.

Aplicações de Pilhas (Stacks)
Função findpath() Continuação...

- A função findpath() utiliza uma pilha para guardar as posições que conformam o caminho. Procura uma posição válida (acima, esquerda, direita, abaixo) e armazena da pilha. Caso não seja possível avançar, remove-se da pilha a última posição.

```
if not self._exitFound(row, col): # Posição atual != Saída
    # Procura a proxima posição valida
    for r in [row-1, row, row+1]:           # Varia a linha
        if not (current_cell.row == s.peek().row and \
                current_cell.col == s.peek().col):
            break
        for c in [col-1, col, col+1]:       # Varia a coluna
            if ((r == row) ^ (c == col)):   # Operador ou exclusivo
                if self._validMove(r,c):        # Se movimento valido
                    self._markPath(r,c)         # Marca como caminho
                    s.push(_CellPosition(r,c)) # Guarda na pilha
                    break
    # Se não encontramos uma posição valida, a celula atual é
    # a mesma da pilha. Então voltar uma posição.
    if current_cell.row == s.peek().row and\
            current_cell.col == s.peek().col:
        not_in_path = s.pop()                   # Remove da pilha
        # Marca a posição como tentativa falha
        self._markTried(not_in_path.row, not_in_path.col)
else:
    return True # Achou o caminho
```

Guarda a posição na pilha.

Remove a última posição da pilha.

Aplicações de Pilhas (Stacks)
Função findpath() Continuação...

- A função findpath() procura a próxima posição válida testando quatro direções possíveis na seguinte ordem: acima, esquerda, direita, abaixo.

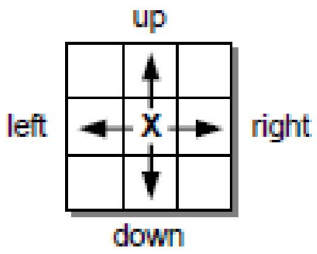


Figure 7.7: The legal moves allowed from a given cell in the maze.

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- O TAD Labirinto (Maze ADT) possui métodos para:
  - i) Determinar se o movimento é valido; ii) Verificar se encontrou a saída;
  - iii) Marca a célula como tentativa falha; iv) Marca a célula como parte do caminho.

Determina se o movimento é valido

Verifica se encontrou a saída

Marca a célula como tentativa falha

Marca a célula como parte do caminho

Celula vazia. Ponteiro nulo.

```
57    # Returns True if the given cell position is a valid move.
      def _validMove( self, row, col ):
59     return row >= 0 and row < self.numRows() \
60        and col >= 0 and col < self.numCols() \
61        and self._mazeCells[row, col] is None
62
63    # Helper method to determine if the exit was found.
      def _exitFound( self, row, col ):
65     return row == self._exitCell.row and \
66            col == self._exitCell.col
67
68    # Drops a "tried" token at the given cell.
      def _markTried( self, row, col ):
70     self._mazeCells.set( row, col, self.TRIED_TOKEN )
71
72    # Drops a "path" token at the given cell.
      def _markPath( self, row, col ):
74     self._mazeCells.set( row, col, self.PATH_TOKEN )
```

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- Finalmente, o TAD Labirinto (Maze ADT) inclui uma classe para armazenar uma coordenada.

```
75
76
77  # Private storage class for holding a cell position.
78  class _CellPosition( object ):
79    def __init__( self, row, col ):
80      self.row = row
81      self.col = col
```

Aplicações de Pilhas (Stacks)
Percurso em um Labirinto – Maze ADT

- Um objeto labirinto (Maze) poderia ser representado da seguinte maneira:

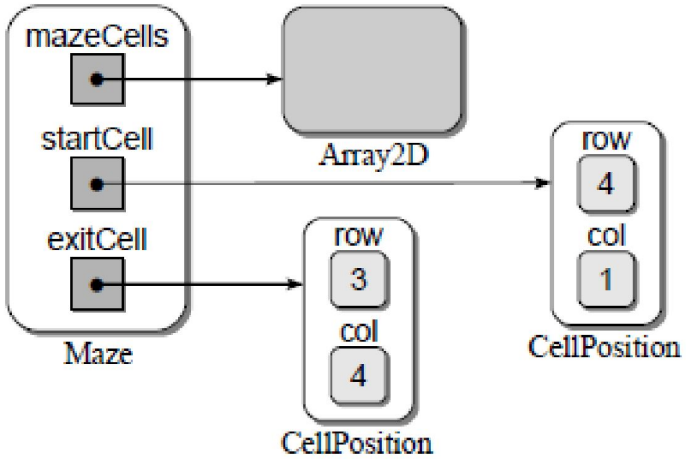


Figure 7.12: A sample Maze ADT object.

Referências

- Rance Necaise. Data Structures and Algorithms Using Python. Capítulo 7. Stacks. 2011.