

UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

PROGRAMA DE FORMACIÓN DUAL

FACULTAD DE INGENIERÍA

REPORTE DE ACTIVIDADES /ALGORITMOS

*PRÁCTICA #1. LABERINTO CON ALGORITMO DE BUSQUEDA DEPTH FIRST SEARCH-PYTHON*

MENTORES:

Gastelum Ossiel

López Becerril Ricardo

Continental Automotive - Querétaro Campus

APRENDICES:

Fuentes Flores Lorena

Martínez Olvera Judith

Vite Gonzales Cynthia

Espinosa Bernal Giovanni

Ugalde Romero Dulce Carolina

Santiago de Querétaro, Qro., 18 de noviembre de 2019.

ÍNDICE

[**I.** **INTRODUCCIÓN** 3](#_Toc24980185)

[**II.** **OBJETIVO** 3](#_Toc24980186)

[**III.** **MARCO TEÓRICO** 4](#_Toc24980187)

[***Breadth-First Search*** 4](#_Toc24980188)

[***Depth-First Search*** 4](#_Toc24980189)

[***Cost-Sensitive Search (A\*)*** 4](#_Toc24980190)

[***Uniform-Cost Search*** 4](#_Toc24980191)

[**IV.** **MATERIALES Y EQUIPO** 5](#_Toc24980192)

[**V.** **METODOLOGÍA** 5](#_Toc24980193)

[**VI.** **RESULTADOS** 6](#_Toc24980194)

[**VII.** **CODIGOS** 6](#_Toc24980195)

[**Código que genera el laberinto:** 6](#_Toc24980196)

[**Código para resolver el laberinto:** 10](#_Toc24980197)

[**VIII.** **CONCLUSIONES** 12](#_Toc24980198)

[**IX.** **DIRECCIÓN GITHUB** 12](#_Toc24980199)

1. **INTRODUCCIÓN**

Un algoritmo de búsqueda está diseñado para localizar un elemento con ciertas características dentro de una estructura de datos; por ejemplo, el registro correspondiente a una persona en una base de datos, identificar el mejor movimiento en un juego, el mejor camino para llegar a un punto en el menor tiempo y con la ruta más óptima, etc.

Los problemas de búsqueda consisten en diferentes características. El espacio de estado es el conjunto de todos los estados posibles donde se puede estar, el estado de inicio es el estado desde donde se comienza la búsqueda y la prueba de objetivo es la función que evalúa el estado actual y verifica si corresponde con el estado objetivo o no. La solución a un problema de búsqueda es una secuencia de acciones que nos lleva del estado inicial al estado objetivo, este plan lo realizan los algoritmos de búsqueda.

1. **OBJETIVO**

Generar un laberinto de dimensiones n x m, así como la solución a este mediante el algoritmo Depth first search

1. **MARCO TEÓRICO**

***Breadth-First Search***

El algoritmo Breadth-First Search comienza considerando todos los caminos posibles ente el punto de inicio y el punto meta, acumula el costo total de cada camino mientras atraviesa cada uno de ellos, esto consume mucho tiempo y requiere de mucha memoria ya que la computadora debe procesar los costos de quizás miles de caminos antes de decidir el óptimo.

***Depth-First Search***

En el algoritmo Depth-First Search un camino es seguido desde el inicio hasta el final y su costo total es calculado, después se sigue otro camino diferente y se calcula su costo, posteriormente ambos costos son comparados y el más costoso es rechazado. Se considera otra ruta y se sigue comparando el costo, continua este proceso hasta que todos los caminos posibles han sido analizados. Esta aproximación minimiza el uso de memoria ya que solo se conserva la ruta más reciente y la menos costosa, el problema con este algoritmo es que puede tomar un largo tiempo de computo especialmente si hay una mala elección del primer camino.

***Cost-Sensitive Search (A\*)***

El algoritmo A\* es una de las mejores y más populares técnicas utilizadas en la búsqueda de rutas y recorridos de gráficos, muchos juegos y mapas basados en la web utilizan este algoritmo para encontrar el camino más costo de manera muy eficiente. Lo que hace A\* es que en cada paso selecciona el nodo de acuerdo con un valor: ' f ', que es un parámetro igual a la suma de otros dos parámetros: ' g ' y ' h '. En cada paso, selecciona el nodo / celda que tiene la ' f ' más baja y procesa ese nodo. ' g ' es el costo de movimiento dese el punto de partida a un nodo siguiente, ' h ' es el costo del movimiento estimado para moverse desde el punto dado hasta el punto meta, este método se conoce como heurístico.

***Uniform-Cost Search***

La búsqueda no informada es una clase de algoritmos de búsqueda de propósito general que opera en forma de fuerza bruta. Los algoritmos de búsqueda no informados no tienen información adicional sobre el estado o el espacio de búsqueda aparte de cómo atravesar el árbol, por lo que también se llama búsqueda ciega.

1. **MATERIALES Y EQUIPO**

* Software Python 3.7

1. **METODOLOGÍA**

La codificación del algoritmo de búsqueda se realizó tomando en cuenta los siguientes pseudocódigos:

def depthFirstSearch(problem):

# crear la estructura stack

fringe = until.Stack()

#Lista para cumular los nodos expandidos

expanded\_nodes = []

#obtener el nodo de inicio y guardarlo.

# Current\_node [0] es el valor del dono actual

# Current\_node [1] almacena una secuencia de pasos para llegar al valor del nodo actual

current\_node = (problem.getStartState (), [])

# While true

while 1:

# Si el nodo actual no ha sido expandido

if not current\_node[0] in expanded\_nodes:

# Añadir el nodo no expandido a la lista de nodos expandidos.

expanded\_nodes.append(current\_node[0])

# Iterar sobre los sucesores del nodo actual

for coordinate, direction, cost in problem.getSuccessors(current\_node[0]):

# Añadir los sucesores al fringe

fringe.push((coordinate, current\_node[1] + [direction]))

# Extraer el primer nodo que se encuentra en la fila fringe

current\_node = fringe.pop()

# Si se ha encontrado la solución, retorna la secuencia de pasos

if problem.isGoalState(current\_node[0])

return current\_node[1]

1. **RESULTADOS**

if \_\_name\_\_ == "\_\_main\_\_":

maze = maze\_generator(3,5)

print\_maze(maze)



Figure 1 Laberinto generado de dimensiones 3 x 5.

if \_\_name\_\_ == "\_\_main\_\_":

maze = maze\_generator(10,30)

print\_maze(maze)

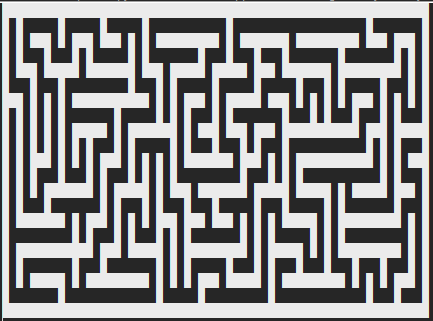


Figure 2 Laberinto generado de dimensiones 10 x 30.

1. **CODIGOS**

**Código que genera el laberinto:**

import random

WALL = "█"

HALL = " "

def absolute\_coordinate(row, col):

abs\_row = (row\*2) + 1

abs\_col = (col\*2) + 1

return (abs\_row, abs\_col)

' ' '

Calcula el número de espacios totales en el vector para generar el laberinto considerando los espacios libres y las paredes de este.

Argumentos:

row (int): coordenada de la fila.

col (int): coordenada de la columna.

Retornos:

abs\_row (int): coordenada absoluta de la fila.

abs\_col (int): coordenada absoluta de la columna.

' ' '

def generate\_raw\_maze(rows, cols):

n\_rows = (rows\*2) + 1

n\_cols = (cols\*2) + 1

maze = [[WALL]\*n\_cols for \_ in range(n\_rows)]

for i in range(rows):

for j in range(cols):

abs\_row, abs\_col = absolute\_coordinate(i,j)

maze[abs\_row][abs\_col] = HALL

return maze

' ' '

Genera la lista de listas con los espacios y las paredes que conforman el laberinto de nxm.

Argumentos:

rows (int): número total de filas.

cols (int): número total de columnas.

Retornos:

maze (list of list).

' ' '

def init\_position():

return [0,0]

' ' '

Define el punto inicial del laberinto siempre será (0,0).

Argumentos:

Ninguno.

Retornos:

maze (list): coordenada de inicio del laberinto (0,0).

' ' '

def init\_state(rows,cols):

current\_state = [[False]\*cols for \_ in range(rows)]

return current\_state

' ' '

Inicia todos los espacios del laberinto como FALSE.

Argumentos:

rows (int): número total de filas.

cols (int): número total de columnas.

Retornos:

current\_state (list of list): lista de todos los espacios del laberinto en FALSE.

' ' '

def get\_successors(row, col, max\_rows, max\_cols):

successors = []

# Get possible successors

possible\_successors = [[row+i,col+j] for i,j in [[0,1],[0,-1],[1,0],[-1,0]]]

for i,j in possible\_successors:

# Check if inside board

if i >= 0 and i < max\_rows:

if j >= 0 and j < max\_cols:

successors.append([i,j])

return successors

' ' '

Obtiene las coordenadas sucesoras de la coordenada actual.

Argumentos:

row (int): coordenada actual de la fila.

col (int): coordenada actual de la columna.

max\_rows (int): número total de filas.

max\_cols (int): número total de columnas.

Retornos:

successors (list): lista de coordenadas de todos los sucesores de la coordenada actual.

' ' '

def is\_goal\_state(state):

is\_goal = all([all(x) for x in state])

return is\_goal

' ' '

Verifica que la lista de estado actual este en TRUE.

Argumentos:

state (list): Lista de las coordenadas visitadas

Retornos:

is\_goal (bool): Retorna TRUE si toda la lista del argumento está en estado TRUE y FALSE si la lista no está completamente en TRUE

' ' '

def remove\_wall(maze, row1, col1, row2, col2):

abs\_coord1 = absolute\_coordinate(row1,col1)

abs\_coord2 = absolute\_coordinate(row2,col2)

if abs\_coord1[0] == abs\_coord2[0]:

row = abs\_coord1[0]

col = min(abs\_coord1[1], abs\_coord2[1]) + 1

elif abs\_coord1[1] == abs\_coord2[1]:

row = min(abs\_coord1[0], abs\_coord2[0]) + 1

col = abs\_coord1[1]

print(row1, col1)

print(row2, col2)

print('~~~~~')

maze[row][col] = HALL

return maze

' ' '

Remueve las paredes y la sustituye por un espacio vacío

Argumentos:

maze (list):

rog1 (int):

col1 (int):

rog2 (int):

col2 (int):

Retornos:

maze (list): Retorna TRUE si toda la lista del argumento está en estado TRUE y FALSE si la lista no está completamente en TRUE

' ' '

def update\_state(state, row, col):

state[row][col] = True

return state

' ' '

Obtiene las coordenadas sucesoras de la coordenada actual.

Argumentos:

row (int): coordenada actual de la fila.

col (int): coordenada actual de la columna.

state (int): número total de filas.

Retornos:

state (list):

' ' '

def print\_maze(maze):

for line in maze:

print(''.join(line))

' ' '

Imprime el laberinto con ayuda de la función .join(line)

Argumentos:

maze (list): lista de espacios y muros que generan el laberinto

Retornos:

Ninguno

' ' '

def maze\_generator(rows, cols):

# Define fringe as an empty list (use append and pop to have a LIFO behavior)

fringe = []

expanded\_coord = []

current\_maze = generate\_raw\_maze(rows, cols)

current\_coord = init\_position() #Posición inicial en 0,0

current\_state = init\_state(rows,cols) #posición inicial en falso todas las posiciones

while True:

if not current\_coord[0] in expanded\_coord:

expanded\_coord.append(current\_coord[0])

# Update state

current\_state = update\_state(current\_state, current\_coord[0][0], current\_coord[0][1])

successors = get\_successors(current\_coord[0][0], current\_coord[0][1], rows, cols)

random.shuffle(successors)

for s in successors:

if not s[0] in expanded\_coord:

fringe.append(s)

if current\_coord[1]:

current\_maze = remove\_wall(current\_maze, current\_coord[0][0], current\_coord[0][1], current\_coord[1][0], current\_coord[1][1])

current\_coord = fringe.pop()

if is\_goal\_state(current\_state):

return current\_maze

' ' '

Obtiene las coordenadas sucesoras de la coordenada actual.

Argumentos:

rows (int): coordenada actual de la fila.

cols (int): coordenada actual de la columna.

Retornos:

current\_maze (list): lista de coordenadas de todos los sucesores de la coordenada actual.

' ' '

if \_\_name\_\_ == "\_\_main\_\_":

maze = maze\_generator(3,3)

print\_maze(maze)

**Código para resolver el laberinto:**

UP = "UP"

DOWN = "DOWN"

LEFT = "LEFT"

RIGHT = "RIGHT"

WALL = "█"

HALL = " "

from maze\_generator import maze\_generator, print\_maze, absolute\_coordinate

def get\_successors(maze, row, col, max\_rows, max\_cols):

successors = []

# Get possible successors

# check left

if inside\_maze(row, col-1, max\_rows, max\_cols):

if wall\_in\_middle(maze, row,col,row,col-1) is False:

successors.append([[row,col-1], [LEFT]])

# check right

if inside\_maze(row, col+1, max\_rows, max\_cols):

if wall\_in\_middle(maze, row,col,row,col+1) is False:

successors.append([[row,col+1], [RIGHT]])

# check up

if inside\_maze(row-1, col, max\_rows, max\_cols):

if wall\_in\_middle(maze, row,col,row-1,col) is False:

successors.append([[row-1,col], [UP]])

# check down

if inside\_maze(row+1, col, max\_rows, max\_cols):

if wall\_in\_middle(maze, row,col,row+1,col) is False:

successors.append([[row+1,col], [DOWN]])

return successors

def wall\_in\_middle(maze, row1, col1, row2, col2):

abs\_coord1 = absolute\_coordinate(row1,col1)

abs\_coord2 = absolute\_coordinate(row2,col2)

if abs\_coord1[0] == abs\_coord2[0]:

row = abs\_coord1[0]

col = min(abs\_coord1[1], abs\_coord2[1]) + 1

elif abs\_coord1[1] == abs\_coord2[1]:

row = min(abs\_coord1[0], abs\_coord2[0]) + 1

col = abs\_coord1[1]

if maze[row][col] == WALL:

return True

return False

def inside\_maze(row, col, max\_rows, max\_cols):

if row >= 0 and row < max\_rows:

if col >= 0 and col < max\_cols:

return True

return False

def init\_position():

#[[init coord], [path to coord]]

pos = [[0,0], []]

return pos

def solve\_maze(maze):

# Define fringe as an empty list (use append and pop to have a LIFO behavior)

fringe = []

expanded\_coord = []

current\_coord = init\_position() #Posición inicial en 0,0

rows = int((len(maze) - 1) / 2)

cols = int((len(maze[0]) - 1) / 2)

goal\_row = rows - 1

goal\_col = cols - 1

while True:

if not current\_coord[0] in expanded\_coord:

expanded\_coord.append(current\_coord[0])

successors = get\_successors(maze, current\_coord[0][0], current\_coord[0][1], rows, cols)

for s in successors:

if not s[0] in expanded\_coord:

s[1] = current\_coord[1] + s[1]

fringe.append(s)

current\_coord = fringe.pop()

if current\_coord[0][0] == goal\_row and current\_coord[0][1] == goal\_col:

return current\_coord[1]

if \_\_name\_\_ == "\_\_main\_\_":

maze = maze\_generator(7,7)

path =solve\_maze(maze)

print\_maze(maze)

print(path)

1. **CONCLUSIONES**
2. **DIRECCIÓN GITHUB**

<https://github.com/DulceCarolina/Control-de-versiones-de-proyecto-formacion-dual/upload/patch-1/Algoritmos>