Universidad del Valle

Ingeniería de Sistemas

Pinocho Univalle

Introducción a la Inteligencia Artificial

Autores:
Andrés Felipe Ruíz Buriticá
Kevin Steven Victoria Ospina
Juan Sebastián González Camacho

3 de mayo de 2023

Índice

1. Introducción

Un agente inteligente, es una entidad capaz de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar un resultado esperado. Es capaz de percibir su medio ambiente con la ayuda de sensores y actuar en ese medio utilizando actuadores.

En este proyecto, implementaremos algunos de los algoritmos de búsqueda no informada, aplicados a los agentes inteligentes, que aprendimos en lo que lleva del curso de Introducción a la Inteligencia Artificial.

2. Objetivo

Este proyecto tiene como objetivo aplicar los conceptos vistos en el transcurso del curso desarrollando un agente inteligente para Pinocho que le ayude a encontrar a Gepetto. Para lograrlo debemos tener en cuenta los siguientes puntos:

- El agente se mueve a los espacios vacíos o hacia Gepetto con costo 1.
- Si el agente pasa por los cigarrillos, cuesta 2.
- Si el agente pasa por algún zorro, cuesta 3.
- La matriz debe leerse desde un archivo de texto.
- Se debe implementar la técnica de amplitud (en este caso aplicando una variante en la que se recorra en zig zag), costo uniforme y profundidad iterativa.
- Debe al final definir con buenos criterios, cuál de las tres estrategias fue mejor y por qué.

3. Desarrollo

Creamos una clase `Maze` para almacenar el tablero de juego, que se puede leer desde una matriz numérica dentro de un archivo de texto. Para esto, hacemos uso de la librería `Numpy`, que nos permite hacer la lectura del archivo de texto y representarlo como un arreglo. Por último, se define el punto de partida (la posición de Pinocho) y el punto de destino (la posición de Gepetto).

```
import numpy as np
from constant import Constant

class Maze(Constant):
    """
    Class that represents a maze.
    """

    def __init__(self, filename:str, matrix = None):
        """
        Initializes the class instance with a numeric
        matrix from a file.

        Args:
            filename (str): path to the file with the
            numeric matrix representing the maze.
        """
        if not matrix is None:
            self.maze = np.array(matrix)
        else:
            # Load maze from file.
            self.maze = np.loadtxt(filename, dtype=int)

        self.matrix = self.maze.tolist()

        # Define the start and goal positions.
        self.start = tuple(
            np.argwhere(self.maze == self.PINOCCHIO)[0])
        self.goal = tuple(
            np.argwhere(self.maze == self.GEPETTO)[0])
```

3.1. Búsqueda por amplitud

El algoritmo de búsqueda por amplitud está contenido dentro de la clase `BFS`, cuyo método constructor recibe únicamente una instancia de `Maze` y a partir de ella define los atributos maze (matriz que representa el laberinto), `start` (Pinocho) y `goal` (Gepetto).

```
from collections import deque

from constant import Constant
from maze import Maze

class BFS(Constant):
    """
    Class that implements the Breadth-Preferring Search
    (BFS) algorithm.
    This algorithm is complete because it always finds
    an answer (if it exists), but it does not guarantee
    that the result is optimal.
    """

    def __init__(self, maze:Maze):
        """
        Initializes the class instance.
        Args:
            maze (Maze): Maze instance that represents
            the board.
        """
        self.maze = maze.maze
        self.start = maze.start
        self.goal = maze.goal
```

La implementación del algoritmo como tal está en el método `solve()` de la misma clase. El método primero inicializa las variables necesarias, entre las cuales se tiene una cola y una lista de nodos visitados. Se toma la posición inicial, donde está Pinocho, y se añade a la cola y a la lista de nodos visitados.

Mientras la cola tenga elementos, se retira el primero de ellos y se evalúa si es la meta (Gepetto) en cuyo caso sale del ciclo. Si no es meta, se explora sus vecinos y se repite el ciclo. Una vez encontrada la meta, se traza el camino desde Gepetto hasta Pinocho.

La lógica del algoritmo está separada en distintos métodos auxiliares que facilitan la lectura del código. Posteriormente se analizará cada método utilizado en la solución.

```
def solve(self) -> list|str:
    """
    Implements the BFS algorithm.This implementation
    avoids returning to already visited nodes.
    Returns:
        (List): path from the start position to the
        end position.
    """
    self._initialize()

    # Enqueue the starting position and mark it as visited.
    self._add_to_queue(self.start)
    self._mark_visited(self.start)

    while self.queue:
        current = self.queue.popleft()
        if self._is_goal(current):
            break
        self._explore_neighbors(current)
    return self._backtrack()
```

El algoritmo alternativo de la búsqueda por amplitud, en donde se empieza buscando de izquierda a derecha y en el siguiente nivel de derecha a izquierda, y viceversa, se implementó en el método `solve_zigzag()`.

Este es muy similar al anterior, solamente que se añade una variable direction (1 o -1) para controlar el sentido de la búsqueda. Así, sí direction es 1, se saca el primer elemento de la cola, si es -1 se saca el último. Cuando se termina de explorar todos los vecinos y si aún hay elementos en la cola, `direction` cambia de signo para comenzar a buscar en el siguiente nivel desde la dirección opuesta.

```
def solve_zigzag(self) -> list|str:
    """
    Implements the BFS algorithm in a zigzag pattern. For
    this, it uses a list and takes out the nodes from the
    extreme left or right depending on the level.
    This implementation avoids returning to already visited
    nodes.
    Returns:
        (List): path from the start position to the end
        position.
    """
    self._initialize()

    # Enqueue the starting position and mark it as visited.
    self._add_to_queue(self.start)
    self._mark_visited(self.start)

    # Initialize a variable to keep track of the direction.
    # 1 for left to right, -1 for right to left.
    direction = 1

    while self.queue:
        if direction == 1:
            current = self.queue.popleft()
        else:
            current = self.queue.pop()
        if self._is_goal(current):
            break
        self._explore_neighbors(current, direction)

        # Change direction if the queue is not empty.
        if self.queue:
            direction *= -1

    return self._backtrack()
```

A continuación, vamos a ver los métodos auxiliares que encapsulan parte de la lógica.

El método `_initialize()` inicializa una cola, un objeto Set (almacena elementos no repetidos) que guardará los nodos visitados, una lista de nodos visitados en orden y un diccionario `parent` que servirá para reconstruir el camino desde la meta hasta el inicio.

```
def _initialize(self):
    """
    Initializes the queue, the visited set and the parent
    dictionary to keep track of the path.
    """
    self.queue = deque()
    self.visited = set()    # disordered, unique
    self.visited_list = []  # ordered
    self.parent = {self.start: None}
```

El método `_is_goal()` toma una posición dentro del laberinto y evalúa si es meta, es decir, si corresponde con Gepetto.

```
def _is_goal(self, position:tuple[int, int]) -> bool:
    """
    Evaluates if a given position is the goal.
    Args:
        position (tuple): a specific position within the
        maze.
    Returns:
        bool: True if the position is the goal.
    """
    return position == self.goal
```

4. Conclusión