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 `Constant`, en la que configuramos a que rol pertenece cada número, de la siguiente manera:

```
class Constant:
    """
    Class that represents the numeric values of the maze.
    """

    WALL = -1
    PINOCCHIO = 0
    EMPTY = 1
    CIGAR = 2
    FOX = 3
    GEPETTO = 4
```

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
```

El método `_explore_neighbors()` toma una posición dentro del laberinto y el sentido de búsqueda (1 de izquierda a derecha, -1 de derecha a izquierda). El método define los posibles movimientos y su orden: arriba, derecha, abajo e izquierda.

Luego, para cada movimiento calcula la próxima posición y si dicha posición es válida (e.g. está dentro de los límites del laberinto) se añade dicha casilla a la lista de nodos visitados, a la cola (según la dirección dada) y al diccionario, indicando que el padre de la próxima posición es el nodo actual.

```
def _explore_neighbors(self,
    current:tuple[int, int], dir:int|None=None):
    """
    Evaluates neighboring cells from a given position based
    on possible moves.
    Args:
        current (tuple): a specific position within the maze.
        dir (int): indicates the search direction.
                    1 or None: from left to right.
                   -1: from right to left.
    """
    # Define the possible moves: Up, Right, Down, Left
    moves = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    for move in moves:
        # Calculates the next position according to the
        # movement.
        next_pos = (current[0] + move[0],
                    current[1] + move[1])

        if self._is_valid_position(next_pos):
            self._mark_visited(next_pos)
            self._add_to_queue(next_pos, dir)
            self._set_parent(next_pos, current)
```

El método `_is_valid_position()` toma una posición (x, y) y evalúa si dicha coordenada está dentro de los límites del laberinto, si no ha sido visitada antes y si no corresponde a un muro. Esas son las condiciones necesarias para considerar una posición válida.

```
def _is_valid_position(self, position:tuple[int, int])->bool:
    """
    Evaluates if the new position is within the bounds of
    the matrix and if the new position hasn't been visited
    and isn't a wall.
    Args:
        position (tuple): a specific position within the
        maze.
    Returns:
        bool: True if the position is valid, that is, it's
        inside the matrix, it hasn't been visited and it's
        not a wall.
    """
    return (
        (0 <= position[0] < self.maze.shape[0])
        and (0 <= position[1] < self.maze.shape[1])
        and position not in self.visited
        and self.maze[position] != self.WALL
    )
```

El método `_mark_visited()` toma una posición (x, y) y la añade al conjunto y a la lista de nodos visitados. En realidad solo es necesario el conjunto visited, pero `visited_list` lo usamos en la interfaz gráfica para ir coloreando los nodos a medida que se visitan.

```
def _mark_visited(self, position:tuple[int, int]):
    """
    Evaluates if a given position has been visited.
    Args:
        position (tuple): a specific position within the
        maze.
    Returns:
        bool: True if the position has been visited.
    """
    self.visited.add(position)
    self.visited_list.append(position)
```

El método `_add_to_queue()` toma una posición (x, y) y una dirección (1 o -1) y añade dicha posición a la cola según la dirección dada. Si la dirección es 1, se añade al final de la cola, si es -1 se añade al principio.

```
def _add_to_queue(self,
    position:tuple[int, int], dir:int|None=None):
    """
    Adds a position to the queue.
    Args:
        position (tuple): a specific position within the
        maze.
        dir (int): indicates the search direction.
                    1 or None: from left to right.
                   -1: from right to left.
    """
    if dir == None or dir == 1:
        self.queue.append(position)
    else:
        self.queue.appendleft(position)
```

El método `_set_parent()` toma dos nodos, primero el hijo y luego el padre, y los añade a un diccionario, siendo la clave el nodo hijo. De manera que podamos consultar en él el nodo padre de cualquier posición.

```
def _set_parent(self,
    child:tuple[int, int], parent:tuple[int, int]):
    """
    Sets a position as parent of another cell in the maze.
    Args:
        child (tuple): a specific position within the maze.
        parent (tuple): a specific position within the maze.
    """
    self.parent[child] = parent
```

Por último, el método `_backtrack()` reconstruye el camino desde la meta (Gepetto) hasta el nodo inicial (Pinocho). Se comienza añadiendo la meta a una lista `path`, luego consultamos el diccionario `parent` para obtener el padre de la meta y añadirlo a `path` hasta llegar al inicio.

Si la meta no está en el diccionario, implica que el laberinto no tiene solución, por lo cual, en dicho caso se mostrará un mensaje indicando que no existe camino entre Pinocho y Gepetto.

```
def _backtrack(self) -> list|str:
    """
    Backtrack from the goal to the start to find the path.
    """
    path = []
    current = self.goal

    while current is not self.start:
        path.append(current)

        # If goal is not a dict key, there is no path from
        start to goal.
        try:
            current = self.parent[current]
        except KeyError:
            return "Doesn't exist solution"

    path.append(self.start)
    #path.reverse()

    return path
```

3.2. Búsqueda por costo uniforme

El algoritmo de búsqueda por costo uniforme está contenido dentro de la clase `UCS`, 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 queue import PriorityQueue

from constant import Constant
from maze import Maze

class UCS(Constant):
    """
    Class that implements the Uniform-Cost Search algorithm.
    This algorithm is complete and optimal, meaning that it
    finds the shortest path if it exists.
    """

    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 de prioridad 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 el elemento con la mayor prioridad (el costo más bajo) 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 Uniform-Cost Search algorithm.
    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, 0)
    self._mark_visited(self.start)

    while not self.queue.empty():
        current_cost, current = self.queue.get()
        if self._is_goal(current):
            break
        self._explore_neighbors(current, current_cost)

    return self._backtrack()
```

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

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

```
def _initialize(self):
    """
    Initializes the queue, the visited set and the parent
    dictionary to keep track of the path.
    """
    self.queue = PriorityQueue()
    self.visited = set()
    self.visited_list = []
    self.parent = {self.start: None}
    self.cost_so_far = {self.start: 0}
```

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:
    """
    Checks if a position is the goal.
    Args:
        position (tuple): a specific position within the
        maze.
    Returns:
        (bool): True if the position is the goal, False
        otherwise.
    """
    return position == self.goal
```

El método `_explore_neighbors()` toma una posición dentro del laberinto y el costo acumulado en ese nodo. El método define los posibles movimientos y su orden: arriba, derecha, abajo e izquierda.

Luego, para cada movimiento calcula la próxima posición y si dicha posición es válida (e.g. está dentro de los límites del laberinto) se calcula el nuevo costo y se añade la tupla (posición, costo acumulado) a la cola de prioridad, se añade el nodo a la lista de visitados y se actualizan los diccionarios parent y `cost_so_far`.

```
def _explore_neighbors(self,
    current:tuple[int, int], current_cost:int):
    """
    Evaluates neighboring cells from a given position based
    on possible moves.
    Args:
        current (tuple): a specific position within the maze.
        current_cost (int): the cost of the current path to
        reach this position.
    """
    # Define the possible moves: Up, Right, Down, Left
    moves = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    for move in moves:
            # Calculates the next position according to the
            movement.
            next_pos = (current[0] + move[0],
                        current[1] + move[1])

            if self._is_valid_position(next_pos):
                # Calculate the cost of the new path and add
                it to the queue.
                new_cost = current_cost + self.maze[next_pos]
                self._add_to_queue(next_pos, new_cost)

                # Mark the cell as visited.
                self._mark_visited(next_pos)

                # Update the parent dictionary and the
                # cost_so_far dictionary.
                self._set_parent(next_pos, current)
                self.cost_so_far[next_pos] = new_cost
```

El método `_is_valid_position()` toma una posición (x, y) y evalúa si dicha coordenada está dentro de los límites del laberinto, si no ha sido visitada antes y si no corresponde a un muro. Esas son las condiciones necesarias para considerar una posición válida.

```
def _is_valid_position(self,
    position:tuple[int, int]) -> bool:
    """
    Checks if a position is within the maze and is not an
    obstacle.
    Args:
    position (tuple): a specific position within the maze.
    Returns:
        (bool): True if the position is valid, False
        otherwise.
    """
    return (
        (0 <= position[0] < self.maze.shape[0])
        and (0 <= position[1] < self.maze.shape[1])
        and position not in self.visited
        and self.maze[position] != self.WALL
    )
```

El método `_mark_visited()` toma una posición (x, y) y la añade al conjunto y a la lista de nodos visitados. En realidad solo es necesario el conjunto `visited`, pero `visited_list` lo usamos en la interfaz gráfica para ir coloreando los nodos a medida que se visitan.

```
def _mark_visited(self, position:tuple[int, int]):
    """
    Marks a position as visited.
    Args:
        position (tuple): a specific position within the
        maze.
    """
    self.visited.add(position)
    self.visited_list.append(position)
```

El método `_add_to_queue()` toma una posición (x, y) y un costo y los añade ambos a la cola.

```
def _add_to_queue(self, position:tuple[int, int], cost:int):
    """
    Adds a position to the queue.
    Args:
        position (tuple): a specific position within the
        maze.
        cost (int): the cost of the current path to reach
        this position.
    """
    self.queue.put((cost, position))
```

El método `_set_parent()` toma dos nodos, primero el hijo y luego el padre, y los añade a un diccionario, siendo la clave el nodo hijo. De manera que podamos consultar en él el nodo padre de cualquier posición.

```
def _set_parent(self,
    child:tuple[int, int], parent:tuple[int, int]):
    """
    Sets the parent of a given position.
    Args:
        child (tuple): a specific position within the maze.
        parent (tuple): a specific position within the maze.
    """
    self.parent[child] = parent
```

Por último, el método `_backtrack()` reconstruye el camino desde la meta (Gepetto) hasta el nodo inicial (Pinocho). Se comienza añadiendo la meta a una lista `path`, luego consultamos el diccionario parent para obtener el padre de la meta y añadirlo a `path` hasta llegar al inicio.

Si la meta no está en el diccionario, implica que el laberinto no tiene solución, por lo cual, en dicho caso se mostrará un mensaje indicando que no existe camino entre Pinocho y Gepetto.

```
def _backtrack(self) -> list|str:
    """
    Backtracks from the goal position to the start position.
    Returns:
    (List): path from the start position to the end position.
    """
    path = []
    current = self.goal

    while current is not self.start:
        path.append(current)

        # If goal is not a dict key, there is no path from
        # start to goal.
        try:
            current = self.parent[current]
        except KeyError:
            return "Doesn't exist a solution"

    path.append(self.start)
    #path.reverse()

    return path
```

3.3. Búsqueda por profundidad iterativa

Aquí va Profundidad Iterativa.

4. Conclusión

Aquí van las conclusiones.