

PROGRAMACIÓN III

TRABAJO PRÁCTICO

Tecnicatura Universitaria en Inteligencia Artificial

ÍNDICE GENERAL

1	INTRODUCCIÓN	2
1.1	Consigna	2
1.2	Pathfinding	2
1.3	La clase Node	6
1.4	La clase Grid	6
1.5	Construcción de soluciones	7
1.6	Estructuras de datos	8
1.7	Laberinto con pesos	12
2	Go RIGHT	14
2.1	Descripción	14
2.2	Implementación	14






INTRODUCCIÓN

Los algoritmos de búsqueda general en espacios de estados son una herramienta fundamental en la resolución de problemas en el ámbito de la inteligencia artificial. Estos algoritmos permiten encontrar soluciones óptimas o subóptimas en un espacio de búsqueda que representa el conjunto de estados y transiciones de un problema.

A lo largo de este trabajo práctico se explorarán diferentes algoritmos de búsqueda en espacios de estados, su implementación en código y su aplicación al problema de escape del laberinto.

1.1 CONSIGNA

Se debe proveer una implementación de los siguientes algoritmos de búsqueda:

-  Búsqueda en anchura (BFS).
-  Búsqueda en profundidad (DFS).
-  Búsqueda de costo uniforme (UCS).
-  Búsqueda avara primero el mejor (GBFS).
-  Búsqueda A*.

Podemos encontrar sus archivos para completar en el directorio `src/pathfinder/search/`

Usaremos para todos ellos su versión de grafo, es decir, manteniendo en memoria los estados ya alcanzados, para evitar caminos redundantes.

1.2 PATHFINDING

Antes de empezar a trabajar, necesitamos familiarizarnos con la herramienta que utilizaremos en este trabajo práctico.

Empecemos por clonar el repositorio:

 Bash

```
git clone https://github.com/maurolucci/tuia-prog3.git  
→ --depth=1
```

A continuación debemos instalar los paquetes requeridos. Para ello ingresamos al repositorio y ejecutamos el siguiente comando:

 Bash

```
pip install -r requirements.txt
```

Ya estamos listos para empezar a trabajar. Ejecutemos el programa y veamos que pasa:

 Bash

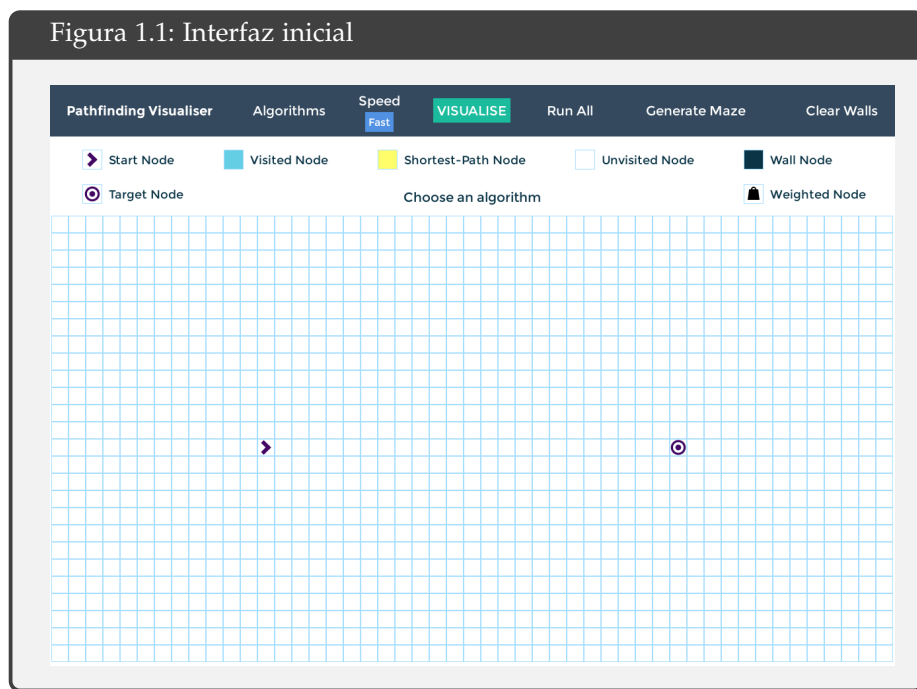
```
python3 run.pyw
```

! Observación

Para poder ejecutar el programa, necesitamos Python 3.10 o superior.

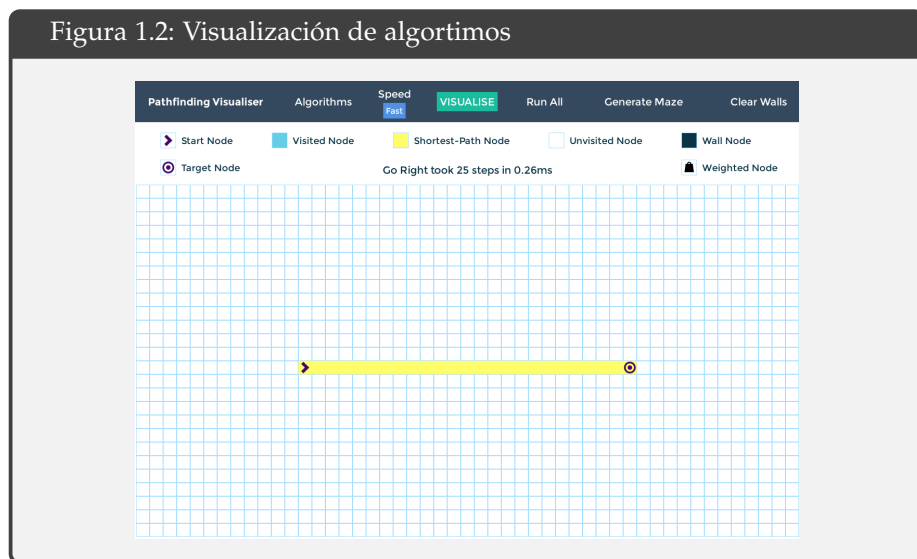
Si hicimos todo bien, deberíamos ver una pantalla como esta:

Figura 1.1: Interfaz inicial



En el menú «Algorithms» tenemos una lista de los algoritmos que vamos a implementar. Elijamos «Go Right» y presionemos «VISUALISE».

Figura 1.2: Visualización de algoritmos

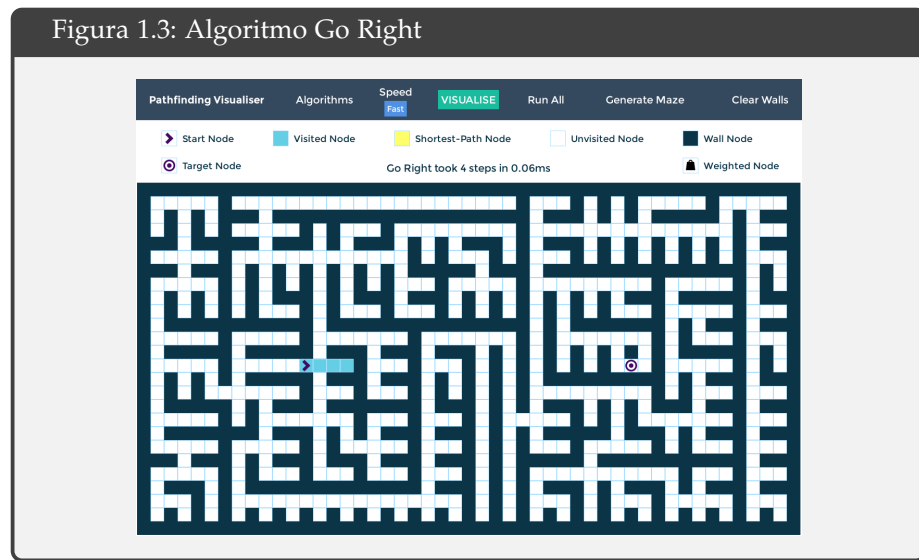


El algoritmo «Go Right» es el único algoritmo que ya viene implementado,

y lo único que hace es ir hacia la derecha. El programa nos permite ver el proceso de exploración de los nodos (en azul) y el camino encontrado (en amarillo).

Probemos ahora un laberinto mas complicado. Podemos dibujar sobre la pantalla haciendo click con el mouse o podemos generar automáticamente uno si vamos al menú «Generate Maze» y elegimos alguno de los algoritmos de generación de laberinto. Volvamos a probar el algoritmo «Go Right».

Figura 1.3: Algoritmo Go Right



Desafortunadamente este algoritmo es demasiado primitivo como para encontrar una ruta.

Notemos que la interfaz nos provee de la opción «Run All» para correr todos los algoritmos (una vez que estén implementados) y comparar los resultados. También podemos cambiar la velocidad de la simulación mediante el menú «Speed».

! Observación

Para poder realizar el trabajo práctico no es necesario leer y entender la totalidad del código del programa; sin embargo hay algunos módulos que nos proveen clases y métodos que nos ayudarán a implementar los diferentes algoritmos de búsqueda. Una explicación de los mismos se presenta a continuación.

1.3 LA CLASE NODE

La clase «Node» representa un nodo del árbol de búsqueda. Dentro de él se almacena el estado (en nuestro caso la posición dentro del laberinto), el costo de camino, el nodo padre y la acción realizada para llegar al estado.

Podemos encontrar la clase «Node» en `src/pathfinder/models`

</> Código

```
class Node:
    def __init__(
        self,
        value: str,
        state: tuple[int, int],
        cost: int,
        parent: Node | None = None,
        action: str | None = None
    ) -> None:
```

1.4 LA CLASE GRID

La clase «Grid» es la clase que se utiliza para representar el laberinto. Nos resultaran de interés los siguientes métodos y atributos:

Podemos encontrar la clase «Grid» en `src/pathfinder/models`

START Este atributo es la posición inicial en el laberinto. Esta representado por una tupla de enteros.

END De forma análoga, el atributo end representa la posición objetivo.

</> Código

```
class Grid:
    def __init__(
        self,
        grid: list[list[Node]],
        start: tuple[int, int],
        end: tuple[int, int]
    ) -> None:
```

GET_COST Al método `get_cost` debemos pasarle una posición dentro del laberinto, y nos devuelve el costo individual de la acción de moverse a esta casilla desde una casilla adyacente.

Más adelante veremos por qué este costo puede ser no unitario.

</> Código

```
def get_cost(self, pos: tuple[int, int]) -> int:
    """Get weight of a node

    Args:
        pos (tuple[int, int]): Cell position

    Returns:
        int: Weight
    """
```

Observación: este método no implementa exactamente la función COSTO-INDIVIDUAL vista en teoría, pero la usaremos con la misma intención.

GET_NEIGHBOURS Para obtener las casillas a las que podemos llegar a partir de una posición usamos el método `get_neighbours`. Nos devuelve un diccionario donde las claves son las direcciones hacia donde podemos ir y los valores son las posiciones a las cuales se llega.

</> Código

```
def get_neighbours(
    self,
    pos: tuple[int, int]
) -> dict[str, tuple[int, int]]:
    """Determine the neighbours of a cell

    Args:
        pos (tuple[int, int]): Cell position

    Returns:
        dict[str, tuple[int, int]]: Action - Position Mapper
    """
```

Observación: combina las funciones ACCIONES y RESULTADO vistas en teoría.

1.5 CONSTRUCCIÓN DE SOLUCIONES

Una vez que nuestro algoritmo haya finalizado, ya sea porque encontró un camino o porque no lo encontró, debemos retornar un objeto.

1.5.1 Solución encontrada

La clase «Solution» es la clase que usamos para construir una solución a partir de un nodo objetivo y un diccionario de estados alcanzados. El método de inicialización de la clase se ocupa de consturir la solución escalando por el atributo parent de cada nodo.

Podemos encontrar la clase «Solution» en `src/pathfinder/models`

</> Código

```
class Solution:
    """Model a solution to a pathfinding problem"""

    def __init__(
        self,
        node: Node,
        reached: dict[tuple[int, int], any],
        time: float = 0
    ) -> None:
```

Observación: el diccionario de alcanzados se utiliza únicamente para pintar en pantalla las casillas alcanzadas con color azul.

1.5.2 Solución no encontrada

Análogamente usamos la clase «NoSolution» cuando no encontramos una solución al problema. Debido a que no hemos llegado al objetivo, no debemos pasarle un nodo.

Podemos encontrar la clase «NoSolution» en `src/pathfinder/models`

</> Código

```
class NoSolution(Solution):
    """Model an empty pathfinding solution"""

    def __init__(
        self,
        reached: dict[tuple[int, int], any],
        time: float = 0
    ) -> None:
```

1.6 ESTRUCTURAS DE DATOS

A medida que vayamos implementando los algoritmos, necesitaremos almacenar los nodos visitados en alguna estructura de datos. Dichas estructuras

Podemos encontrar la clase «Frontier» en `src/pathfinder/models`

heredan todas de la clase «Frontier». En ella se definen los siguientes métodos de interés:

INIT Es el constructor que nos permite crear una frontera vacía.

</> Código

```
class Frontier:
    """Model a frontier for managing nodes"""

    def __init__(self) -> None:
```

ADD El método add recibe un nodo y nos permite agregar un elemento a la frontera.

</> Código

```
def add(self, node: Node) -> None:
    """Add a new node to the frontier

    Args:
        node (Node): Maze node
    """
```

IS_EMPTY Usamos el método is_empty cuando queremos verificar si la frontera se encuentra vacía.

</> Código

```
def is_empty(self) -> bool:
    """Check if the frontier is empty

    Returns:
        bool: Whether the frontier is empty
    """
```

! Observación

Si bien la clase Frontier define la interfaz común a las diferentes estructuras de datos, nunca la utilizaremos directamente, sino a través de las clases que heredan de ella.

1.6.1 Pila

La clase `StackFrontier` se comporta como una pila y solo agrega un método a la interfaz:

REMOVE Eliminamos el elemento correspondiente al orden de extracción utilizando el método `remove`, el cual nos devuelve el nodo eliminado de la frontera.

</> Código

```
class StackFrontier(Frontier):
    def remove(self) -> Node:
        """Remove element from the stack

        Raises:
            Exception: Empty Frontier

        Returns:
            Node: Cell (Node) in a matrix
        """
```

! Observación

El orden de extracción de una pila está dado por la política «*LIFO*»: el último en entrar, es el primero en salir.

1.6.2 Cola

Si queremos utilizar una cola usamos la clase `QueueFrontier`. Al igual que la pila, también agrega el mismo método a la interfaz.

REMOVE Eliminamos el elemento correspondiente al orden de extracción utilizando el método `remove`, el cual nos devuelve el nodo eliminado de la frontera.

</> Código

```
class QueueFrontier(Frontier):
    def remove(self) -> Node:
        """Remove element from the queue

        Raises:
            Exception: Empty Frontier

        Returns:
            Node: Cell (Node) in a matrix
        """
```

! Observación

El orden de extracción de una cola está dado por la política «FIFO»: el primero en entrar, es el primero en salir.

1.6.3 Cola de prioridad

Finalmente también disponemos de una cola de prioridad implementada en la clase `PriorityQueueFrontier`. Los métodos de interés para esta clase son:

ADD El método `add` definido en esta clase reemplaza al de la clase padre. En este caso podemos pasar un argumento opcional para indicar la prioridad del elemento en cuestión.

</> Código

```
def add(self, node: Node, priority: int = 0) -> None:
    """Add a new node into the frontier

    Args:
        node (AStarNode): Maze node
        priority (int, optional): Node priority.
    """
```

POP Eliminamos el elemento correspondiente al orden de extracción utilizando el método `pop`, el cual nos devuelve el nodo eliminado de la frontera.

</> Código

```
def pop(self) -> Node:
    """Remove a node from the frontier

    Returns:
        AStarNode: Node to be removed
    """
```

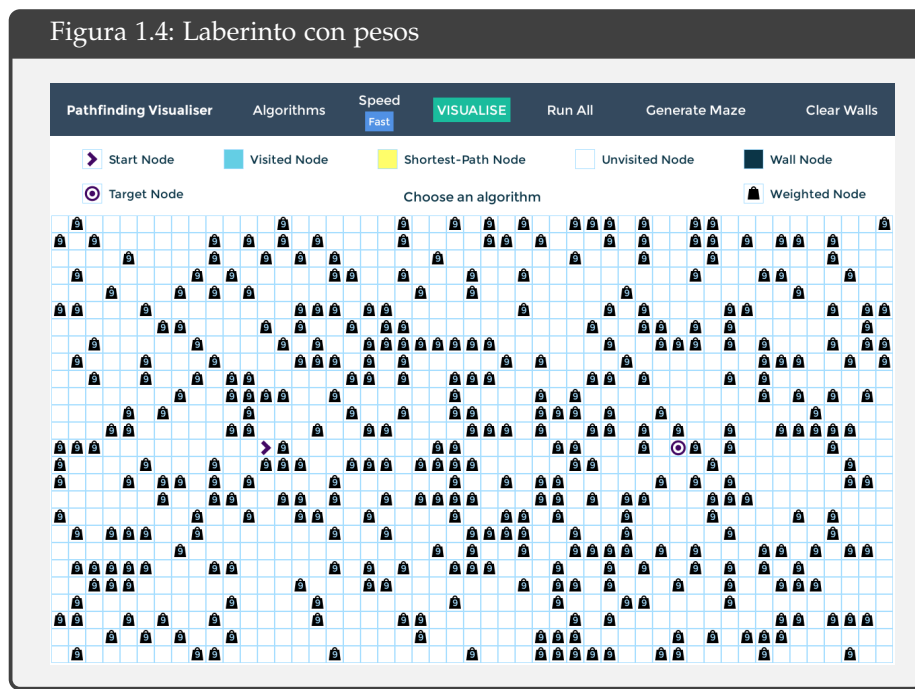
! Observación

El orden de extracción de esta estructura está dado por la prioridad con la que se agregaron los elementos: el primero en ser removido es el elemento de menor prioridad.

1.7 LABERINTO CON PESOS

Para probar los algoritmos de búsqueda que permiten costos no unitarios, debemos crear un laberinto con pesos, donde las paredes se pueden cruzar pero pagando un costo asociado. Podemos generar un laberinto automáticamente eligiendo la opción del menú «Generate Maze» y luego «Basic Weight Maze»; o bien manteniendo presionada una tecla numérica y luego dibujando con el click del mouse.

Figura 1.4: Laberinto con pesos



Go RIGHT

2.1 DESCRIPCIÓN

El algoritmo «Go Right» es simplemente un modelo para ejemplificar como deben implementarse el resto de los algoritmos. La forma en la que explora los nodos es simplemente fijándose si puede avanzar hacia la derecha.

Podemos encontrar el algoritmo «GoRight» en `src/pathfinder/search`

2.2 IMPLEMENTACIÓN

Observemos su implementación para comprender cómo fue programado.

1. Empezamos por crear el nodo inicial. Observemos que hacemos esto partiendo del estado inicial y no nos ha costado nada llegar hasta allí.

</> Código

```
# Initialize a node with the initial position
node = Node("", state=grid.start, cost=0,
            parent=None, action=None)
```

2. Luego, necesitamos crear un diccionario en donde almacenamos los estados alcanzados. En un principio, el único estado alcanzado es el inicial.

</> Código

```
# Initialize the reached dictionary with
# the initial state
reached = {}
reached[node.state] = True
```

Observación: usamos un diccionario y no un conjunto, para que posteriormente se pueda visualizar de forma correcta el orden en que se pintan de azul las casillas alcanzadas.

3. Realizamos el test objetivo.

```
</> Código

# Return if the node contains a goal state
if node.state == grid.end:
    return Solution(node, reached)
```

4. A continuación, elegimos una estructura de datos adecuada para almacenar los nodos de la frontera y agregamos la raíz.

```
</> Código

# Initialize the frontier with the initial node
# In this example, the frontier is a queue
frontier = QueueFrontier()
frontier.add(node)
```

5. Una vez que terminamos estas tareas de inicialización, estamos dispuestos a comenzar el algoritmo con un bucle `while`. Si no encontramos ningún nodo para extraer de la frontera, quiere decir que no hemos encontrado una solución al problema.

```
</> Código

while True:

    # Fail if the frontier is empty
    if frontier.is_empty():
        return NoSolution(reached)
```

6. De lo contrario (si restan nodos por expandir), extraemos el primero de ellos.

```
</> Código

# Remove a node from the frontier
node = frontier.remove()
```

7. Generamos todos los posibles estados sucesores.

```
</> Código

# Go right
successors = grid.get_neighbours(node.state)
```


8. Puesto que nuestro algoritmo de ejemplo solo va hacia la derecha, verificamos que esto sea posible.

```
</> Código  
  
if 'right' in successors:
```

9. En caso afirmativo, debemos expandir el nodo. Para ello, obtenemos el estado sucesor.

```
</> Código  
  
# Get the successor  
new_state = successors['right']
```

10. Verificamos que el estado sucesor no haya sido alcanzado.

```
</> Código  
  
# Check if the successor is not reached  
if new_state not in reached:
```

11. En caso afirmativo, generamos el nodo hijo. Para ello, construimos un nodo con el estado sucesor, su costo de camino (que es el costo de camino de su padre más el costo individual de moverse al estado sucesor), su nodo padre y la acción que lo generó (que en este caso siempre es right).

```
</> Código  
  
# Initialize the son node  
new_node = Node("", new_state,  
                node.cost + grid.get_cost(new_state),  
                parent=node, action='right')
```

12. Marcamos al sucesor como alcanzado.

```
</> Código  
  
# Mark the successor as reached  
reached[new_state] = True
```

13. Realizamos el test objetivo. En este caso, el test objetivo se ejecuta inmediatamente antes de agregar un nuevo nodo a la frontera.

</> Código

```
# Return if the node contains a goal state  
# In this example, the goal test is run  
# before adding a new node to the frontier  
if new_state == grid.end:  
    return Solution(new_node, reached)
```

14. Finalmente ya estamos listos para agregar el nodo a la frontera.

</> Código

```
# Add the new node to the frontier  
frontier.add(new_node)
```