

# PROGRAMACIÓN III

## TRABAJO PRÁCTICO

Tecnicatura Universitaria en Inteligencia Artificial

---

## ÍNDICE GENERAL

---

<b>I</b>	<b>CONCEPTOS PREVIOS</b>	<b>2</b>
1	INTRODUCCIÓN	3
1.1	Pathfinding . . . . .	3
1.2	La clase Node . . . . .	6
1.3	La clase Grid . . . . .	6
1.4	Construcción de soluciones . . . . .	8
1.5	Estructuras de datos . . . . .	9
2	GO RIGHT	13
2.1	Descripción . . . . .	13
2.2	Implementación . . . . .	13
<b>II</b>	<b>ALGORITMOS DE BÚSQUEDA</b>	<b>16</b>
3	BFS	17
4	DFS	18
5	UCS	19
6	GREEDY	21
7	A*	22

## Parte I

### CONCEPTOS PREVIOS

---

## 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 búsqueda de ruta.


En particular, se abordarán los siguientes algoritmos:

- W Búsqueda en anchura (BFS).
- W Búsqueda en profundidad (DFS).
- W Búsqueda de costo uniforme (UCS).
- Búsqueda avariciosa (primero el mejor).
- W Búsqueda A\*.

### 1.1 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:

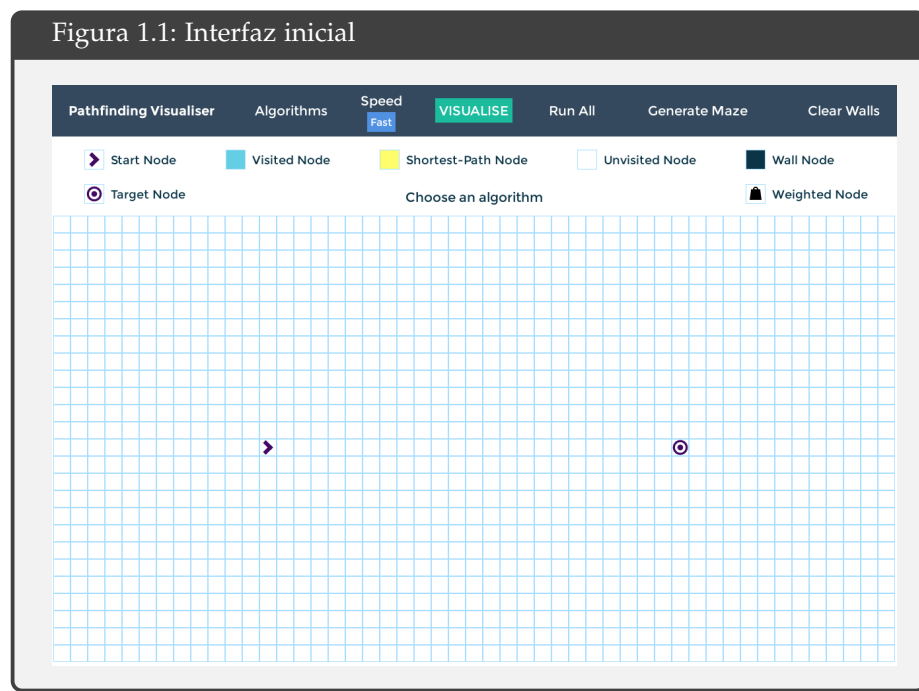
```
Bash  
python3 run.pyw
```

### ! Observación

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

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

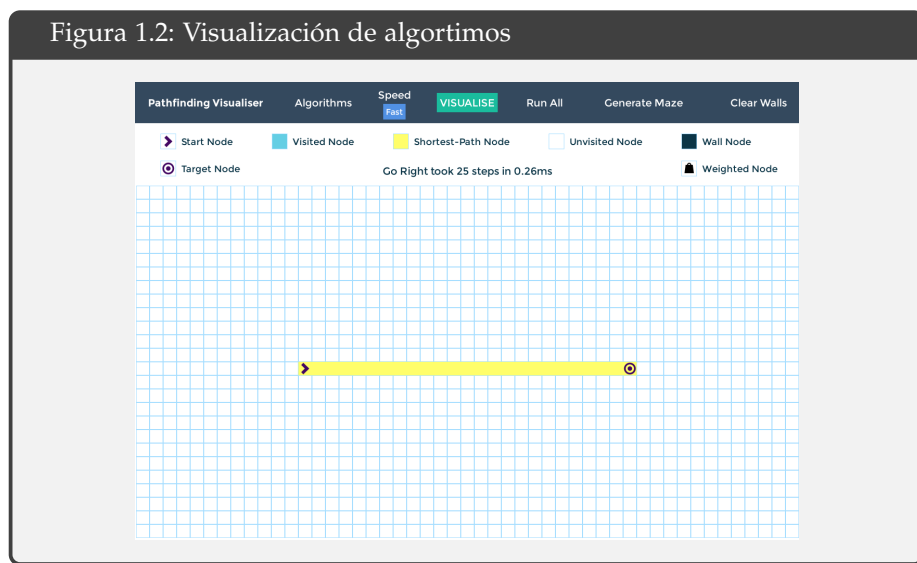
Figura 1.1: Interfaz inicial



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

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

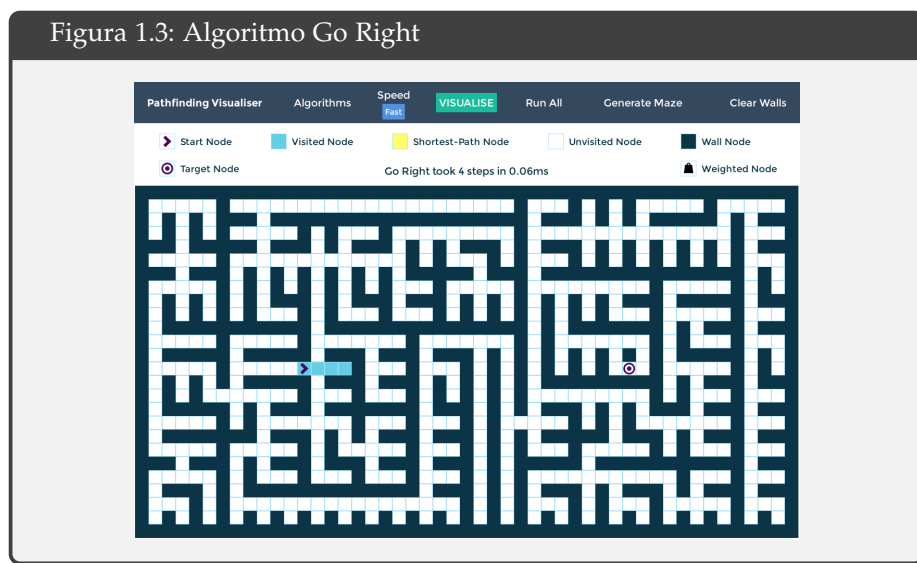
Figura 1.2: Visualización de algoritmos



proceso de exploración de los nodos, y el camino encontrado.

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.

### 1.2 LA CLASE NODE

La clase «Node» nos permite almacenar un nodo dentro de un grafo de búsqueda. Dentro de el 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:
```

### 1.3 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.

&lt;/&gt; 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.*

&lt;/&gt; 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
    """
```

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.

&lt;/&gt; 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
    """
```



## 1.4 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.

### 1.4.1 Solución encontrada

La clase «Solution» es la clase que se usamos para construir una respuesta válida al problema. La construimos a partir del nodo final y un diccionario de posiciones exploradas.

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

</> Código

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

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

### 1.4.2 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 la posición final.

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

</> Código

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

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

## 1.5 ESTRUCTURAS DE DATOS

A medida que vayamos implementando los algoritmos, necesitaremos almacenar los nodos visitados en alguna estructura de datos. Dichas estructuras heredan todas de la clase «Frontier». En ella se definen los siguientes métodos de interés:

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

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

```
</> Código

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
    """
```

**CONTAINS\_STATE** Para comprobar si un estado ya se encuentra en algún nodo de la frontera usamos el método contains\_state. Debemos pasarle una posición (tupla de enteros) y nos devuelve un booleano.

```
</> Código

def contains_state(self, state: tuple[int, int]) -> bool:
    """Check if a state exists in the frontier

    Args:
        state (tuple[int, int]): Position of a node

    Returns:
        bool: Whether the provided state exists
    """
```

**IS\_EMPTY** Usamos el método is\_empty cuando queremos verificar si la frontera se encuentra vacía.

&lt;/&gt; 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.

## 1.5.1 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.

&lt;/&gt; 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.

### 1.5.2 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.

### 1.5.3 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.

---

## 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 como 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)
```

2. Luego necesitamos crear un diccionario en donde posteriormente almacenaremos los estados alcanzados.

</> Código

```
# Initialize the explored dictionary to be empty
explored = {}
```

3. A continuación elegimos una estructura de datos adecuada para almacenar los nodos generados, y agregamos el primero de ellos.

</> Código

```
# Initialize the frontier with the initial node
frontier = QueueFrontier()
frontier.add(node)
```

4. Una vez que terminamos estas tareas de inicialización, estamos dispuestos a comenzar el algoritmo con un bucle **while**.

- a) Si no encontramos ningún nodo para explorar en nuestra estructura, quiere decir que no hemos encontrado una solución al problema.

</> Código

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

- b) De lo contrario (si restan nodos por explorar) extraemos el primero de ellos.

</> Código

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

- c) Lo marcamos como ya explorado.

</> Código

```
# Mark the node as explored
explored[node.state] = True
```

- d) Verificamos si el estado del nodo elegido es el estado objetivo, en cuyo caso devolvemos la solución al problema.

</> Código

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

- e) De no ser así (no hemos llegado a la solución) debemos generar todos los posibles estados sucesores.

```
</> Código  
  
# Go right  
neighbours = grid.get_neighbours(node.state)
```

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

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

- g) En caso afirmativo debemos agregar el nodo a nuestra estructura. Para ello:

- i) Almacenamos el estado de destino.

```
</> Código  
  
new_state = neighbours['right']
```

- ii) Construimos un nodo con dicho estado y actualizamos el costo de llegar hasta el.

```
</> Código  
  
new_node = Node("", new_state, \  
node.cost + grid.get_cost(new_state))
```

- iii) Establecemos el nodo padre.

```
</> Código  
  
new_node.parent = node
```

- iv) Guardamos la acción realizada para llegar al estado destino.

```
</> Código  
  
new_node.action = 'right'
```

- v) Finalmente ya estamos listos para agregar el nodo a la estructura.

```
</> Código  
  
# Add the new node to the frontier  
frontier.add(new_node)
```



## Parte II

### ALGORITMOS DE BÚSQUEDA

Los algoritmos de búsqueda que se deben implementar se mencionan en las siguientes secciones. Usaremos para todos ellos su versión en grafos, es decir, manteniendo en memoria los estados ya alcanzados, para evitar caminos redundantes.

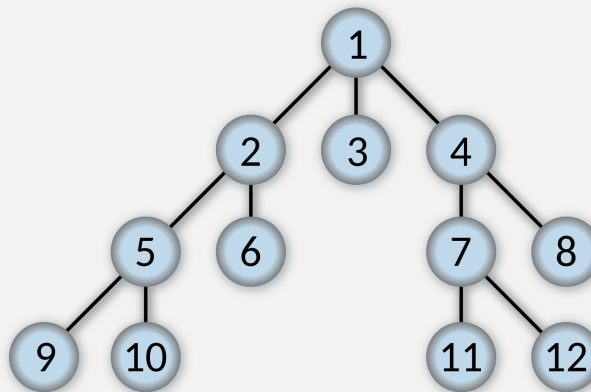
## BFS

El algoritmo de búsqueda BFS (Breadth-First Search, por sus siglas en inglés) es un algoritmo de búsqueda en grafos que se utiliza para recorrer y buscar todos los nodos de un grafo en un orden sistemático y estructurado.

El algoritmo comienza explorando el nodo inicial, y luego explora todos los vecinos del nodo inicial en orden de cercanía. Después de visitar todos los vecinos del nodo inicial, el algoritmo continúa explorando todos los vecinos de los nodos visitados en orden de cercanía. En otras palabras, el algoritmo explora primero los nodos cercanos al nodo inicial antes de continuar con los nodos más alejados.

*Podemos implementar el algoritmo «BFS» dentro de la carpeta `src` en `/pathfinder/search/bfs.py`*

Figura 3.1: Orden de búsqueda en BFS



### ! Observación

Para poder implementar el algoritmo, es importante comprender el orden de exploración de los nodos. ¿Que estructura de datos tiene un orden de extracción idéntico al que necesitamos?

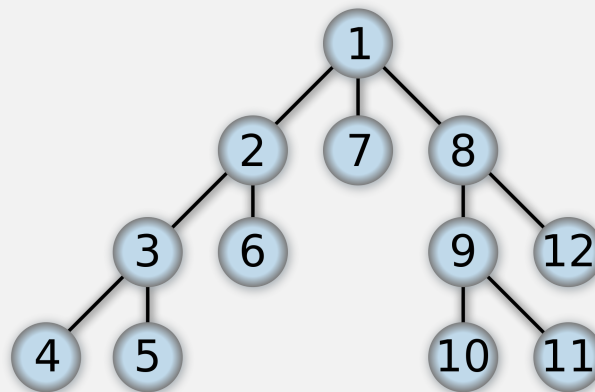
## DFS

El algoritmo DFS (Depth-First Search, por sus siglas en inglés) es otro algoritmo de búsqueda en grafos utilizado para recorrer y buscar todos los nodos de un grafo. A diferencia del algoritmo BFS, el algoritmo DFS explora los nodos en profundidad.

El algoritmo comienza explorando un nodo inicial y luego genera todos los sucesores del nodo en un orden arbitrario. Una vez que se visita un nodo, el algoritmo se mueve a uno de los sucesores no visitados y continúa explorando a partir de allí. Si un nodo no tiene sucesores no visitados, el algoritmo retrocede al nodo anterior y explora otros nodos no visitados desde allí.

*Podemos implementar el algoritmo «DFS» dentro de la carpeta `src` en `/pathfinder/search/dfs.py`*

Figura 4.1: Orden de búsqueda en DFS



### ! Observación

Al igual que en DFS, existe una estructura de datos que tiene un orden de extracción ideal para DFS. ¿Cuál es?

## UCS

El algoritmo UCS (Uniform Cost Search, por sus siglas en inglés) es un algoritmo de búsqueda en grafos que utiliza el costo acumulado del camino para seleccionar el siguiente nodo a expandir. Este algoritmo es similar al algoritmo de Dijkstra, que también utiliza el costo acumulado de la ruta para encontrar la ruta más corta entre dos nodos en un grafo con pesos.

*Podemos implementar el algoritmo «UCS» dentro de la carpeta `src` en `/pathfinder/search/ucs.py`*

### ! Observación

Vale la pena notar que el algoritmo de Dijkstra encuentra el camino más corto desde un vértice a todos los otros vértices del grafo, mientras que UCS encuentra el camino más corto del estado inicial al estado objetivo más cercano.

El algoritmo UCS comienza en un nodo inicial y expande los nodos vecinos del nodo actual en orden ascendente de costo acumulado de la ruta.

Recordemos el algoritmo de Dijkstra:

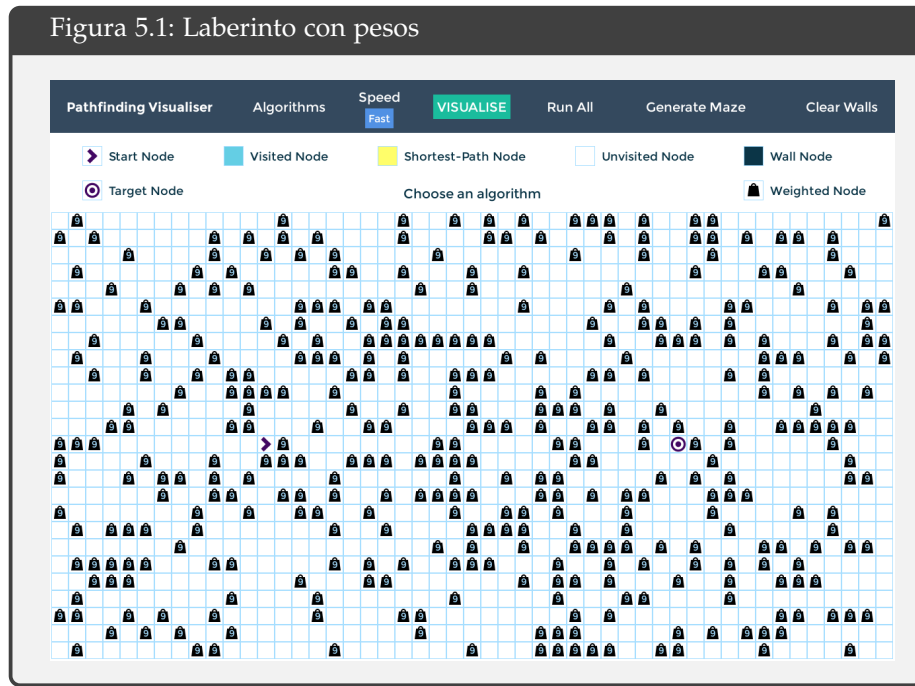
*Nuevamente resulta fundamental identificar la estructura de datos a utilizar en este algoritmo.*

1. Primero, notemos que la ruta más corta del vértice inicial ( $a$ ) al vértice  $a$  es la ruta de cero aristas, que tiene peso 0. Así, inicializamos  $L(a) = 0$ .
2. No sabemos aún el valor de una ruta más corta de  $a$  a los otros vértices, entonces para cada vértice  $v \neq a$ , inicializamos  $L(v) = \infty$ .
3. Inicializamos el conjunto  $T$  como el conjunto de todos los vértices, es decir:  $T = V$ .
4. Seleccionamos un vértice  $v \in T$  tal que  $L(v)$  sea mínimo.
5. Quitamos el vértice  $v$  del conjunto  $T$ :  $T = T - \{v\}$ .
6. Para cada  $w \in T$  adyacente a  $v$ , actualizamos su etiqueta:  $L(w) = \min\{L(w), L(v) + c(v, w)\}$ .
7. Si  $z \in T$ , repetimos desde el paso 4, si no, hemos terminado y  $L(z)$  es el valor de la ruta más corta entre  $a$  y  $z$ .

## 5. UCS

Para probar el algoritmo UCS 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 5.1: Laberinto con pesos



---

## GREEDY

---

El algoritmo Greedy de búsqueda en grafos es un algoritmo de búsqueda informada que utiliza una heurística para decidir qué nodo explorar a continuación. Este algoritmo se basa en la idea de que, en cada paso, el nodo más prometedor para explorar es aquel que parece más cercano al objetivo en términos de la heurística utilizada.

El algoritmo Greedy comienza en un nodo inicial y evalúa los nodos vecinos del nodo actual en función de la heurística utilizada. Luego, selecciona el nodo vecino que parece más cercano al objetivo y lo expande. Este proceso se repite hasta que el algoritmo llega al nodo objetivo o hasta que no hay más nodos por explorar.

*Podemos  
implementar el  
algoritmo  
«Greedy» dentro  
de la carpeta `src`  
en  
`/pathfinder/search/gbfs.py`*

---

## A\*

---

El algoritmo A\* es un algoritmo de búsqueda informada que combina la heurística del algoritmo Greedy con la información del costo real del camino para encontrar la solución óptima en un grafo con pesos. Este algoritmo utiliza una función de evaluación  $f(n) = g(n) + h'(n)$  para seleccionar el siguiente nodo a expandir, donde  $g(n)$  es el costo acumulado del camino desde el nodo inicial hasta el nodo actual, y  $h'(n)$  es la estimación del costo desde el nodo actual hasta el nodo objetivo utilizando una heurística.

*Podemos implementar el algoritmo «A\*» dentro de la carpeta `src` en `/pathfinder/search/astar.py`*

### ! Observación

La Búsqueda de Costo Uniforme es un caso particular del algoritmo de búsqueda A\* si la heurística de este último es una función constante.