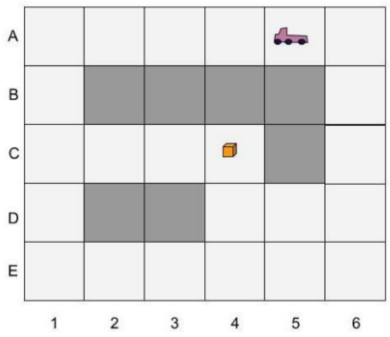
Asignatura	Nombres
Razonamiento y Planificación Automática: Resolución de problema mediante búsqueda heurística	**Joel Orellana - Lyn Mantilla - Ruben Aponte**

Se parte de un problema de búsqueda, probando los diferentes algoritmos propuestos ***Breadth First Search*** (BFS) o Amplitud, ***Depth First Search*** (DFS) o Profundidad y **A*** con las heurísticas ***Euclídea*** y ***Manhattan***. El problema se puede ver en la Figura 1.

Figura 1. Mapa de búsqueda del problema planteado.



Es interesante el enfoque empírico y el análisis de casos particulares.

Muchos desarrollos manuales de los árboles.

Falta en parte razonar sobre las propiedades generales de los algoritmos, por lo que no se puede evaluar si las conocéis:

- Si profundidad es un algoritmo óptimo
- Si las heurísticas son admisibles
- Si A* es óptimo o no con estas heurísticas
- A* reduce el número de nodos expandidos con respecto a Amplitud porque no necesita explorar tantos caminos
- Diferencia entre las dos heurísticas

Recomiendo que reviséis la clase de resolución para ver algunas afirmaciones teóricas que se pueden hacer (desde las propiedades conocidas de los algoritmos).

Tareas a realizar

Probar a resolver el problema con distintos algoritmos de búsqueda: amplitud considerando lo siguiente:

1. El **coste real** (g) del movimiento de la furgoneta es 1 por casilla.

La librería searchai presenta unos costos por defecto definidos en el método SearchProblem (Figura 2).

Figura 2. Función de costo de la clase SearchProblem.

```
def cost(self, state, action, state2):
'''Returns the cost of applying `action` from `state` to `
   The returned value is a number (integer or floating point).
   By default this function returns `1`.
'''
    return 1
```

Véase que, por defecto, todas las acciones tienen un costo de 1.

En el programa analizado, se ha modificado el método para que los costos y acciones sean obtenidos directamente de un diccionario *COSTS* (Figura 3).

2. Para A*, deberá probar como función heurística la distancia de **Manhattan**, y la distancia **Euclídea**. Dentro de la clase GameWalkPuzzle se encuentra el método para definir la heurística del algoritmo A* (Figura 4).

Observe que para definir la heurística del problema de búsqueda se utilizan las coordenadas (x, y) del estado en el que se encuentra el camión y la posición de la meta. Dado que la heurística es una estimación de cuanto falta por recorrer hasta la meta, la distancia Euclídea sería la más exacta, sin embargo, el camión tiene la restricción de no poder moverse en diagonales, lo que implica que la distancia de Manhattan, que solo considera las diferencias absolutas de movimientos en x y y, podría ser más beneficiosa al problema dado.

Figura 3. Diccionario con los costos para cada movimiento.

```
COSTS = {
    "up": 1.0,
    "down": 1.0,
    "right": 1.0,
    "left": 1.0,
}
```

Figura 4. Método heurístico de la clase GameWalkPuzzle.

```
def heuristic(self, state):
    x, y = state
    gx, gy = self.goal
    # Distancia Euclidea
    return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)
    # Distancia de Manhattan
    # return abs(x - gx) + abs(y - gy)
```

3. La **entrada** del código deberán ser **ficheros de texto o matrices codificadas como se indica anteriormente**

Esto se puede lograr a través de una implementación con *open(), close()* y *read()* en Python, utilizando la función *with*.

La función descrita (ver Figura 5), permite leer los mapas en un archivo texto que se encuentre en el mismo nivel del archivo de código fuente.

4. La **salida** será una secuencia de movimientos a realizar para alcanzar el estado objetivo utilizando una notación específica. Además, debe mostrar el **número de nodos expandidos**, y **el coste total** (suma de los costes de las acciones) de la solución.

Se implementa a través de la función *resultado_experimento* y además tiene una salida en consola con la información solicitada.

Cada punto representa el movimiento de T hasta llegar a P. Se muestra el costo total de la solución (*Total cost of solution*) , el número de estados para llegar a la solución (*Total length of solution*), el máximo tamaño de la frontera (región límite entre nodos no visitados y visitados, *max fringe size*), el total de nodos visitados (*visited nodes*) y el número de veces que se ejecutó el algoritmo hasta llegar a la solución (*iterations*).

5. Deberá probarse en las siguientes circunstancias, y **comparar los resultados numéricos y la solución obtenida**:

    5.a.) El estado inicial de la figura.

Teniendo en cuenta lo descrito en los puntos anteriores, se procede a realizar el despliegue de cada uno de los algoritmos, encontrando los resultados de la Figura 7 para el problema dado.

La expansión de nodos para cada uno de los algoritmos en diagrama de árbol se explicará con mayor detalle (explicando la variación de costes, heurísticas y nodos a expandir según cada algoritmo) en el punto 5c.

5.b.) Un estado inicial en el que el algoritmo de búsqueda en profundidad obtenga la solución óptima expandiendo menos nodos que el resto.

Figura 5. Abrir mapa desde un archivo de texto externo.

```
def read_map(file=None):
    try:
        with open('maze.txt') as f:
            print('Se ha detectado el siguiente mapa:')
            MAP = f.read()
            print(MAP)
    except FileNotFoundError:
        print('No se dectectado un archivo válido maze.txt')
        print('Se procederá a trabajar con el mapa por defecto:')
        MAP = """
##############################
#         #              #   #
# ####    ########       #   #
#  T #    #              #   #
#    ###     ####   ######   #
#         ####      #        #
#            #  #   #   #### #
#     ######    #       # P  #
#        #      #           #
##############################
"""
        print(MAP)
    return [list(x) for x in MAP.split("\n") if x]
```

Figura 6. Resultado en consola para uno de los algoritmos de búsqueda (BFS).

```
Breadth-First Search (Amplitud):
########
#    T·#
# ####·#
#   P#·#
# ##···#
#      #
########
Total length of solution: 8
Total cost of solution: 7.0
max fringe size: 4
visited nodes: 17
iterations: 17
```

Para este caso, se analizan los posibles escenarios en los cuales puede iniciar el camión y la ubicación del paquete en el mapa. El mapa cuenta con 23 casillas disponibles para movimientos. Teniendo en cuenta el camión y el paquete, podrían existir 23 x 22 = 506 escenarios distintos. Se programa una rutina en Python para obtener los 506 escenarios y evaluar cada uno de los algoritmos y comparar sus métricas (Obtenidas del método *getTotalCost* del método *searchInfo* y de *user_viewer.stats*).

Figura 7. Resultado para los 4 algoritmos de búsqueda.

```
Breadth-First Search (Amplitud):
########
#    T·#
# ####·#
#   P#·#
# ##···#
#      #
########
Total length of solution: 8
Total cost of solution: 7.0
max fringe size: 4
visited nodes: 17
iterations: 17

Depth-First Search (Profundidad):
########
#····T #
#·#### #
#···P# #
# ##   #
#      #
########
Total length of solution: 10
Total cost of solution: 9.0
max fringe size: 3
visited nodes: 10
iterations: 10

A* Heuristics (Euclidean):
########
#    T·#
# ####·#
#   P#·#
# ##···#
#      #
########
Total length of solution: 8
Total cost of solution: 7.0
max fringe size: 5
visited nodes: 11
iterations: 11

A* Heuristics (Manhattan):
########
#    T·#
# ####·#
#   P#·#
# ##···#
#      #
########
Total length of solution: 8
Total cost of solution: 7.0
max fringe size: 5
visited nodes: 11
iterations: 11
```

Figura 8. Generación de todos los posibles escenarios del mapa dado.

	Map	BFS Cost	DFS Cost	A*(E) Cost	A*(M) Cost	BFS Visited Nodes	DFS Visited Nodes	A*(E) Visited Nodes	A*(M) Visited Nodes
1	Map1	1.0	1.0	1.0	1.0	3	2	2	2
2	Map2	2.0	2.0	2.0	2.0	5	3	3	3
3	Map3	3.0	3.0	3.0	3.0	7	4	4	4
4	Map4	4.0	4.0	4.0	4.0	9	5	5	5
5	Map5	5.0	5.0	5.0	5.0	13	6	6	6
...	...	...	...	...	...	...	...	...	...
502	Map502	5.0	5.0	5.0	5.0	16	6	6	6
503	Map503	4.0	10.0	4.0	4.0	11	11	5	5
504	Map504	3.0	11.0	3.0	3.0	9	12	4	4
505	Map505	2.0	16.0	2.0	2.0	7	21	3	3
506	Map506	1.0	1.0	1.0	1.0	3	22	2	2

506 rows × 9 columns

Filtrando los resultados para identificar los escenarios que minimizan el costo y los nodos visitados para el algoritmo DFS, se encuentra que existen 52 escenarios que llevan el costo a 1.0, de los cuales, además, 23 cuentan con el mínimo valor de nodos expandidos (2 nodos). Estos 23 escenarios comparten estas mismas características:

- El camión se encuentra justo al lado del paquete. Esto minimiza el costo (costo = 1.0).
- Teniendo en cuenta cómo se apilan los movimientos del DFS, es decir, $['up','down','right','left']$ (de acuerdo a cómo fue programado por *simpleai*), solo los escenarios en donde se restringe el primer movimiento hacia el paquete son los que minimizan los nodos expandidos.

En ninguno de los 23 escenarios, el costo mínimo para DFS fue menor que para BFS o para A* en sus dos heurísticas, sólo logró igualarlos. Con respecto a la cantidad de nodos expandidos, sólo fueron inferiores comparados con el BFS (nodos DFS = 2 vs. Nodos BFS = 3 o 4) pero no con el A* en sus dos heurísticas, a los cuales sólo pudo igualar.

Analizando los movimientos para cada uno de los algoritmos para un caso particular, se puede ver como se expanden los nodos en la Figura 9. Los 23 escenarios se pueden ver en la Figura 10.

Figura 9. Representación gráfica de expansión de nodos para un escenario ejemplo (Mapa47).

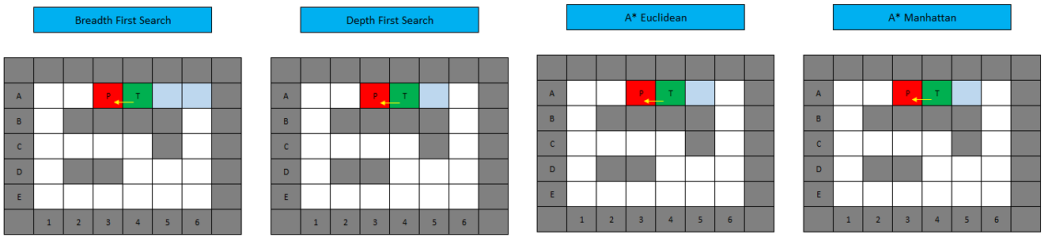
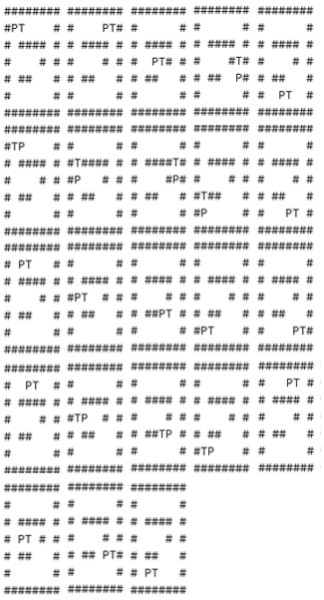


Figura 10. Escenarios que minimizan el costo y nodos expandidos para DFS.



	index	Map	BFS Cost	DFS Cost	A*(E) Cost	A*(M) Cost	BFS Visited Nodes	DFS Visited Nodes	A*(E) Visited Nodes	A*(M) Visited Nodes
0	1	Map1	1.0	1.0	1.0	1.0	3	2	2	2
1	23	Map23	1.0	1.0	1.0	1.0	3	2	2	2
2	24	Map24	1.0	1.0	1.0	1.0	3	2	2	2
3	47	Map47	1.0	1.0	1.0	1.0	3	2	2	2
4	70	Map70	1.0	1.0	1.0	1.0	3	2	2	2
5	93	Map93	1.0	1.0	1.0	1.0	3	2	2	2
6	183	Map183	1.0	1.0	1.0	1.0	3	2	2	2
7	185	Map185	1.0	1.0	1.0	1.0	3	2	2	2
8	207	Map207	1.0	1.0	1.0	1.0	4	2	2	2
9	208	Map208	1.0	1.0	1.0	1.0	3	2	2	2
10	231	Map231	1.0	1.0	1.0	1.0	3	2	2	2
11	272	Map272	1.0	1.0	1.0	1.0	3	2	2	2
12	323	Map323	1.0	1.0	1.0	1.0	4	2	2	2
13	345	Map345	1.0	1.0	1.0	1.0	4	2	2	2
14	346	Map346	1.0	1.0	1.0	1.0	4	2	2	2
15	365	Map365	1.0	1.0	1.0	1.0	3	2	2	2
16	388	Map388	1.0	1.0	1.0	1.0	3	2	2	2
17	392	Map392	1.0	1.0	1.0	1.0	3	2	2	2
18	414	Map414	1.0	1.0	1.0	1.0	3	2	2	2
19	415	Map415	1.0	1.0	1.0	1.0	3	2	2	2
20	438	Map438	1.0	1.0	1.0	1.0	4	2	2	2
21	461	Map461	1.0	1.0	1.0	1.0	4	2	2	2
22	484	Map484	1.0	1.0	1.0	1.0	3	2	2	2

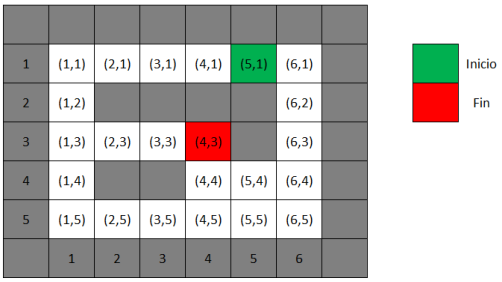
5.c.) Una situación en la que el coste del movimiento varía de la siguiente forma: los movimientos hacia **abajo**, **izquierda** y **derecha** tienen un coste de 1, mientras que los movimientos hacia **arriba** tienen un coste de **5**.

Primero, actualizamos el diccionario de costos de la Figura 2. Cuando T tiene acciones disponibles para poder moverse, el algoritmo sigue el orden definido por la lista que retorna el método *actions* de la clase *GameWalkPuzzle*; dichas acciones pueden modificarse junto a su costo en el diccionario *COSTS*. Por ejemplo, si T se encuentra en un estado en el que puede desplazarse a cualquier dirección, la lista que se retorna es: $['up','down','right','left']$

Además, la implementación de los algoritmos de búsqueda (DFS, BFS y A*) de la librería *simpleai* tienen el parámetro *graph_search=True*, que restringe la posibilidad de regresar a estados ya visitados.

Para entender mejor cómo funcionan los distintos costos, partiremos desde un diagrama que contiene las posiciones de las casillas con en la Figura 11.

Figura 11. Diagrama de posiciones del mapa con las coordenadas x y y.



A partir del diagrama de posiciones, generamos un árbol que contiene los posibles movimientos usando primero la búsqueda no informada BFS (*Breadth First Search, búsqueda en amplitud*), ésta no

toma en cuenta el costo para llegar al destino, sino más bien, busca el estado final hasta llegar a él. El árbol BFS se presenta en la Figura 12.

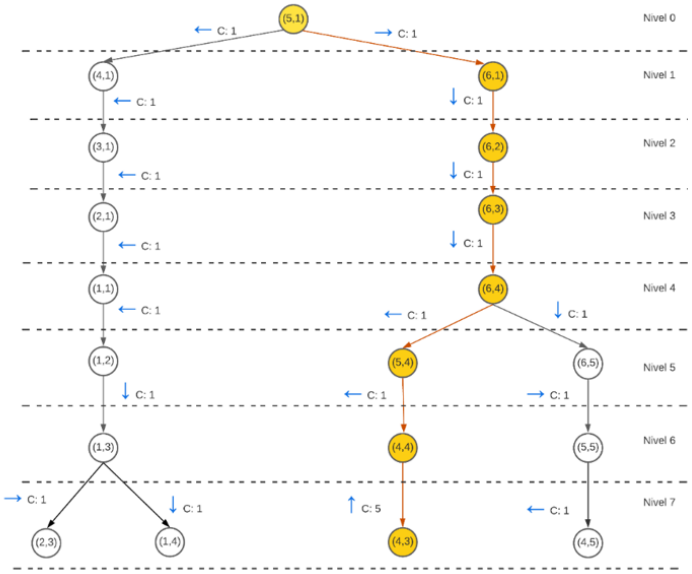
Las ramas del árbol se expandirán desde la raíz a cada uno de los nodos nivel por nivel en la profundidad del árbol, *simpleai* ocupa la librería *deque* para implementar una cola FIFO (*primero en entrar, primero en salir*). El orden en el que se ingresan cada uno de los nodos a la cola, importa para poder elegir cual es el siguiente nodo por visitar. El algoritmo entonces se implementa así :

BFS:

Posición inicial: (5,1) Ya que la posición inicial no es el nodo meta, el algoritmo analiza las acciones disponibles desde el estado. Acciones disponibles: [‘right’, ‘left’]. Se ingresan a la cola las posiciones respectivas a los movimientos disponibles: [(6,1), (4,1)]. Se visita el nodo en la posición (6,1) por ser el primer elemento de la cola. El nodo (6,1) no es la meta, por lo que se elimina de la cola y se agregan sus hijos, en el orden de los movimientos disponibles: [‘down’] -> (6,2). La cola ahora está formada por: [(4,1), (6,2)]. El algoritmo repite la secuencia anterior tomando el primer elemento de la cola, analizando sí es la meta, y agregando los nodos en el orden de las acciones disponibles hasta llegar a la meta.

Figura 12. Árbol y resultados para BFS.

```
Breadth-First Search (Amplitud):
########
#    T·#
# ####·#
#   P#·#
# ##···#
#      #
########
Total length of solution: 8
Total cost of solution: 11.0
max fringe size: 4
visited nodes: 17
iterations: 17
```



Analizando resultados:

El costo de cada segmento hasta llegar a la meta no será tomado en cuenta para encontrar la ruta a la misma. Por lo que BFS no buscará la ruta de costo mínimo. Los estados (nodos) que forman la ruta de inicio a fin son los siguientes:

$$(5,1) \rightarrow (6,1) \rightarrow (6,2) \rightarrow (6,3) \rightarrow (6,4) \rightarrow (5,4) \rightarrow (4,4) \rightarrow (4,3)$$

En total, 8 nodos visitados, los mismos que los mostrados en el *total length of solution* de la salida del programa. Al sumar los costos de los nodos de la ruta, encontramos que: $C_{total} = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 5 = 11.0$, mostrando el mismo costo de la salida del programa en *total cost of solution.*

El número de nodos visitados es el total de nodos que ingresan a la cola en todo el recorrido de inicio a fin:

$$\begin{bmatrix}(5,1),(6,1)(4,1),(6,2),(3,1),(6,3),(2,1),(6,4),(1,1),(6,5),(5,4),(1,2),(5,5),\\(4,4),(1,3),(4,5),(4,3)\end{bmatrix}$$

El total de nodos visitados: 17, los mismos que el programa presenta en la salida *visited nodes*. El máximo número de nodos por visitar ocurre en el nivel 6 del árbol, en el que se tienen los nodos pendientes $(2,3),(1,4),(4,3),(4,5)$, el mismo resultado se presenta en la salida del programa *max fringe size.* Y el número de iteraciones es 17, las mismas que el número de nodos visitados, ya que cada nodo agregado a la cola implica una iteración más del algoritmo de búsqueda.

DFS:

El algoritmo DFS tampoco tomará en cuenta sí un camino tiene mayor costo que los demás, en este caso, DFS utiliza una pila(stack) con la técnica LIFO (Last In, First Out), el orden de nodos en la pila es influenciado por el orden de movimientos disponibles entregados por el método actions, de esta forma, DFS tomará una rama del árbol hasta llegar a la meta o a un nodo hoja. El árbol formado por el problema utilizando DFS se muestra en la Figura 13.

El algoritmo DFS entonces se implementa así:

Posición inicial: (5,1) Ya que la posición inicial no es el nodo meta, el algoritmo analiza las acciones disponibles desde el estado. Acciones disponibles: [‘right’, ‘left’]. Siguiendo el orden LIFO, se ingresan a la pila las posiciones respectivas a los movimientos disponibles: $\begin{bmatrix}(4,1)\\(6,1)\end{bmatrix}$

Se visita el nodo en la posición (4,1) por ser el último elemento de la pila.

El nodo (4,1) no es la meta, por lo que se elimina de la pila y se agregan sus hijos, en el orden de los movimientos disponibles: [‘left’] -> (3,1). La pila ahora está formada por: $\begin{bmatrix}(3,1)\\(6,1)\end{bmatrix}$

El algoritmo repite la secuencia anterior tomando el último elemento en entrar a la pila, analizando sí es la meta, y agregando los nodos a la pila en el orden de las acciones disponibles hasta llegar a la meta. La salida del programa muestra los resultados en la Figura 13:

Analizando resultados:

Total length of solution: Es el número de estados(nodos) de la ruta de inicio a fin, en DFS la ruta viene indicada por:

$$(5,1) \rightarrow (4,1) \rightarrow (3,1) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)$$

En total, 10 estados, los mismos que se presentan a la salida del programa.

Total cost of solution: Aunque DFS tampoco busca la ruta de menor coste, ninguno de los estados anteriores involucra el movimiento *“up”*, por lo que el costo total será de $C_{total} = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9.0$ el mismo resultado se presenta a la salida del programa.

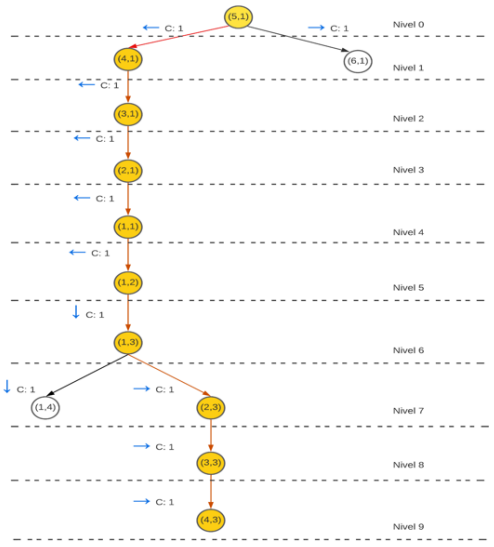
Max fringe size: Los nodos máximos por visitar, ocurre en el nivel 7 del árbol, cuando se deben visitar los nodos $(1,4), (2,3)$ y aún se tiene en memoria el nodo $(6,1)$, en total, 3 nodos pendientes por visitar.

visited nodes: Puesto que DFS explora todos los nodos siguiendo una rama hasta llegar a un nodo hoja o a la meta, los nodos visitados actualmente en la ruta serán los mismos del *total length of solution*, en total, 10.

iterations: El número de veces que se emplea el algoritmo DFS, las mismas del total de nodos visitados: 10.

Figura 13. Árbol y resultado para DFS.

```
Depth-First Search (Profundidad):
########
#····T #
#·#### #
#···P# #
# ##   #
#      #
########
Total length of solution: 10
Total cost of solution: 9.0
max fringe size: 3
visited nodes: 10
iterations: 10
```



Búsqueda A*

Este algoritmo de búsqueda informada busca la ruta de costo mínimo para avanzar a través de la función de costo $f(n) = c(n) + h(n)$, donde $c(n)$ es el costo para llegar desde el inicio hasta el nodo n y $h(n)$ es la función heurística, una suposición estimada de lo que falta para llegar hasta la meta. El árbol formado por el problema utilizado se muestra en la Figura 14.

La heurística utilizada en el ejercicio es de dos tipos, como fue especificada en 5a). Las heurísticas utilizadas no alteran el orden de la expansión de los nodos, por lo que el árbol presentado a continuación será el mismo para ambas distancias (Euclidiana y Manhattan).

Cálculo de $h(n)$: Los cálculos de las heurísticas no son influenciados por el costo del camino, simplemente son estimaciones de la distancia al destino, por ejemplo, para el nodo inicial (5,1) se presentan a continuación las heurísticas respectivas basadas en las respectivas fórmulas de distancia Euclídea y de Manhattan:

Posición: (5,1) Meta: (4,3)

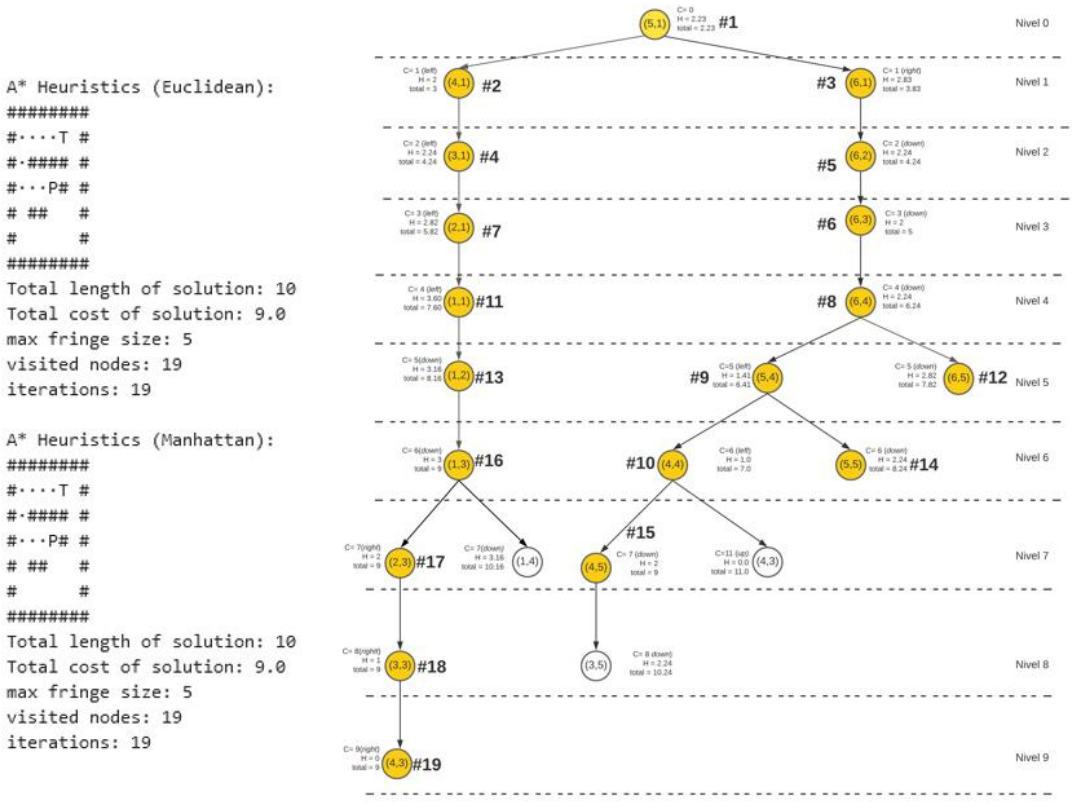
$$d_{euclidea} = \sqrt{(x - x_f)^2 + (y - y_f)^2} = \sqrt{(5-4)^2 + (1-3)^2} = 2.23 = h(n)$$

$$d_{Manhattan} = |x - x_f| + |y - y_f| = |5-4| + |1-3| = 3 = h(n)$$

El mapa de árbol usando ambas heurísticas será el mismo dado que la estimación del costo de cada movimiento no se toma en cuenta en $h(n)$. La Figura 14 muestra el árbol usando heurística con distancia Euclídea.

A* evaluará en cada nodo $f(n)$, comprobará sí el nodo es la meta, en caso de no serlo, expandirá la frontera a sus vecinos para calcular los costos, de todos los nodos en la frontera, tomará el de menor $f(n)$ para verificar sí es el nodo meta, sí no lo es, lo expande, agregando todos sus nodos hijos a la frontera y así sucesivamente. Cabe mencionar que A* no necesariamente tomará como solución la primera vez que una ruta llegue al destino. Seguirá probando hasta agotar las posibles rutas y de ellas tomará la de menor costo.

Figura 14. Árbol y resultado para A.*



La Figura 14 muestra el orden en el que se van seleccionando los nodos de la frontera en base a $f(n)$ mínimo. Observe el efecto que produce el costo de 5.0 (*'up'*) en el proceso de selección de nodos, el costo total de esa ruta se eleva a 11.0 por lo que el algoritmo sigue buscando nuevas rutas alternativas de costo menor hasta llegar a la meta.

Analizando resultados:

Total length of solution: Sí consideramos solamente los nodos que conectan inicio-meta en la ruta de menor costo, encontramos que son los estados:

$$(5,1) \rightarrow (4,1) \rightarrow (3,1) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)$$

En total 10, los mismos que presenta el programa.

Total cost of solution: La ruta final tiene un costo de 9.0 según se observa en la Figura 14, puede deducirse que $f(n) = c(n) + h(n) = 9.0 + 0.0 = 9.0$ el resultado anterior es el mismo que presenta la salida del programa.

max fringe size: El número máximo de nodos por explorar se produce cuando se visita el nodo (1,1), en ese momento, los nodos en frontera son: $(6,5), (1,2), (5,5), (4,5), (4,3)$ En total, 5.

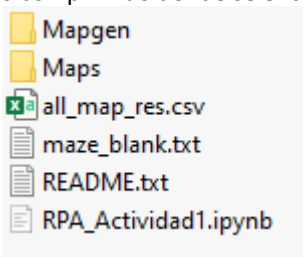
visited nodes: La cantidad de nodos que A* ha visitado, no necesariamente son los mismos de la ruta final que toma, en la figura se numeran los nodos visitados, en total 19.

iterations: El número de veces que se utiliza A* para buscar la meta, en este caso, las mismas veces que los nodos visitados: 19.

ANEXOS

Como parte de este documento, se cuenta con el cuaderno o *notebook* de Jupyter con el código en lenguaje Python y que sustenta el ejercicio realizado, el cual fue generado a partir del *script* entregado. Este, se encuentra en un archivo comprimido zip "RPA_Actividad_1.zip", junto con los mapas generados (ver Figura 15).

Figura 15. Archivo comprimido donde se encuentra el código.



BIBLIOGRAFÍA

Hurbans, R. (2020). *Grokking Artificial Intelligence Algorithms*. Manning Publications.

Joshi, P. (2017). *Artificial intelligence with python*. Packt Publishing Ltd.

Russell, S. J., & Norvig, P. (2004). *Inteligencia Artificial: un enfoque moderno* (Issues 04; Q335, R8y 2004.).