Introducción a la inteligencia artificial

Juan Sebastián Rodríguez Trujillo – Código: 1310272

Escuela de Ingeniería de Sistemas y Computación, Universidad del Valle
Santiago de Cali, Colombia

juan.sebastian.rodriguez@correounivalle.edu.co

Resumen: El siguiente informe tiene como propósito plantear algunos algoritmos de búsqueda ya sean básicos (búsqueda no informada) o heurísticos (búsqueda informada) e implementarlos para el desarrollo de un problema de inteligencia artificial.

I. INTRODUCCION

Las técnicas de búsqueda son una serie de esquemas de representación del conocimiento, que mediante diversos algoritmos nos permite resolver ciertos problemas desde el punto de vista de la inteligencia artificial, dichas búsquedas pueden ser no informadas como por ejemplo la búsqueda por amplitud, costo uniforme y profundidad o en su defecto pueden ser informadas como por ejemplo la búsqueda avara y estrella.

II. PLANTEAMIENTO DEL PROBLEMA

La Universidad del Valle ha construido un prototipo de Robot llamado Ubícame en UV, cuya tarea es ubicar a las personas cuando desean buscar algún profesor o sitio de la Universidad, el Robot tiene información del ambiente que consiste en los obstáculos que hay, su ubicación inicial y su objetivo. El ambiente es representado con una matriz de enteros de tamaño nxn como se muestra en la **(Imagen 1)**:

```
10,
1,1,1,1,1,1,1,1,1,1,
1,S,0,0,1,1,0,0,0,1,
1,0,0,0,1,1,0,0,0,1,
1,0,0,0,0,0,0,0,0,1,
1,1,1,0,1,1,C,1,1,1,
1,1,1,0,1,1,C,1,1,1,
1,0,0,0,0,0,0,0,0,1,
1,0,0,0,1,1,0,0,0,1,
1,0,0,0,1,1,0,0,M,1,
1,1,1,1,1,1,1,1,1,1,
```

(Imagen 1)

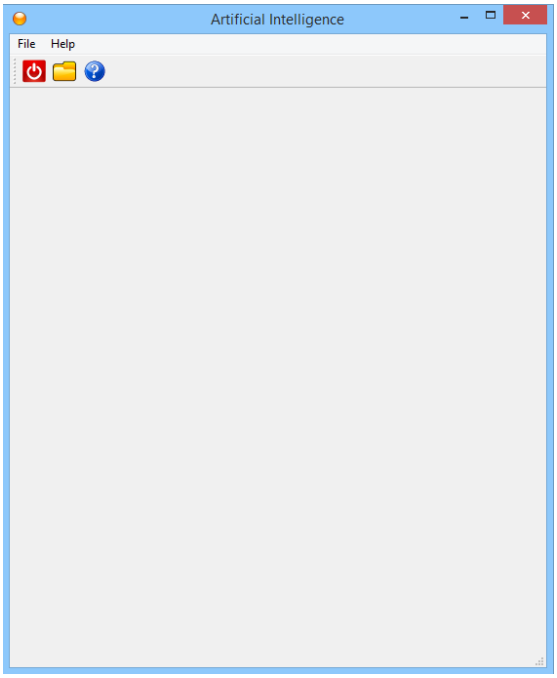
En la **(Imagen 1)** se muestra un ejemplo del formato de la entrada de datos que manipula la aplicación, en cuyo caso la primera línea contiene el tamaño de la maño de la matriz, seguido por la matriz misma que representa el laberinto por el cual se aplicaran las diversas búsquedas, el formato de esta matriz consta de una serie de String cuya representación se muestra en la **(Imagen 2)**:

S	**Inicio**
M	**Meta**
C	**Costo**
0	**Camino**
1	**Pared**

(Imagen 2)

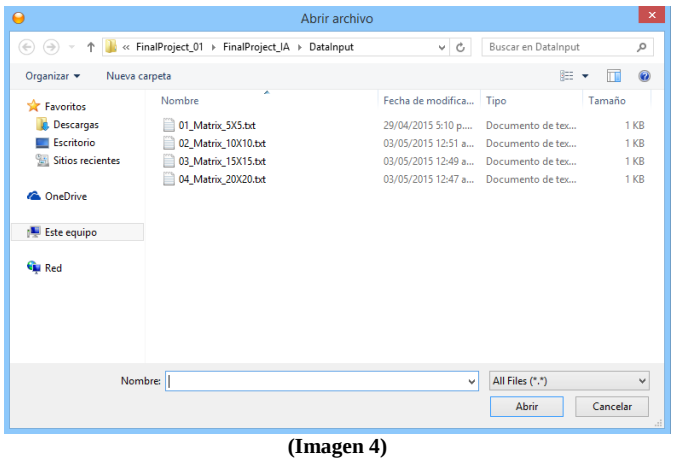
III. DEASARROLLO DEL PROBLEMA

La aplicación fue desarrollada en el lenguaje de programación Python Versión 3.4, implementa la biblioteca QT4 Versión 4.11 para generar su GUI y PyOpenGL Versión 3.1 para generar su diseño gráfico. En la **(Imagen 3)** se muestra la interfaz gráfica que se visualiza al iniciar la aplicación:



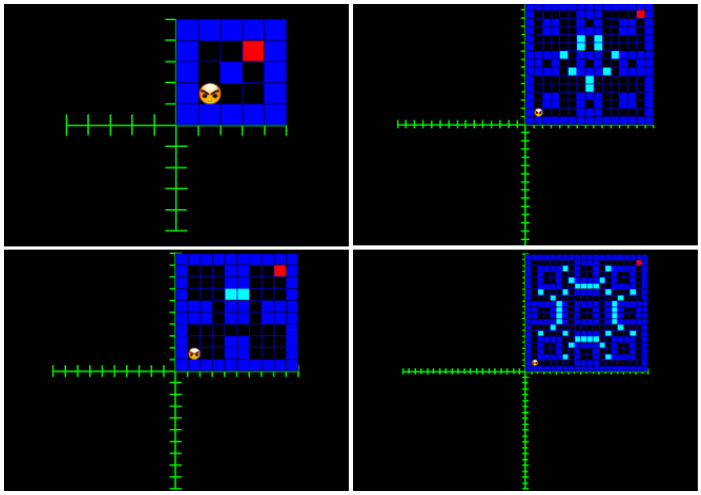
(Imagen 3)

En la barra de herramienta se encontrara el botón (Open File) el cual permitirá abrir los archivos disponibles para ejecutar las diferentes búsquedas:



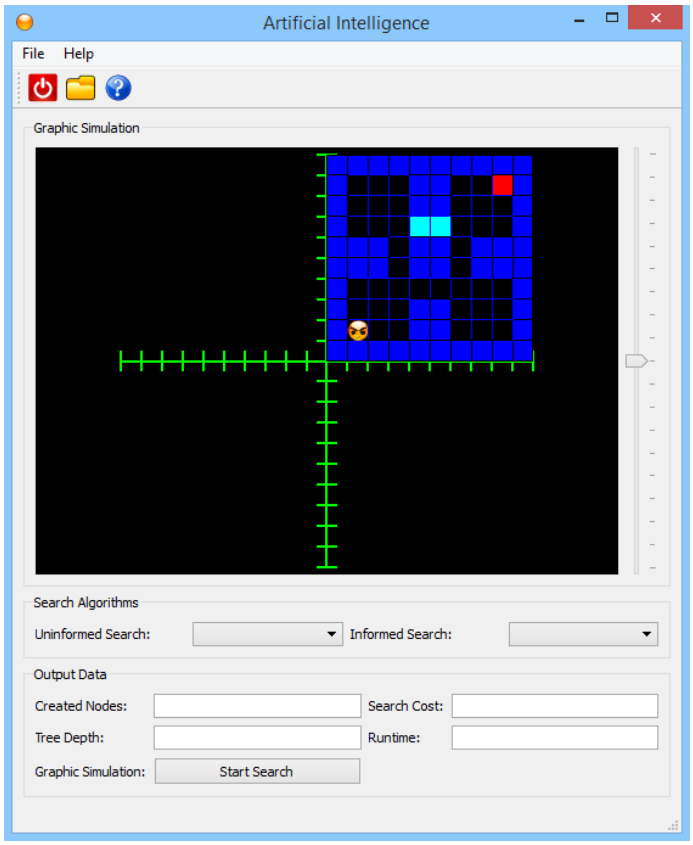
(Imagen 4)

Al cargar uno de los archivos de texto, se podrá visualizar diversos diseños de laberintos en los cuales se encontrara un pequeño personaje cuyo objetivo es llegar al punto de meta:



(Imagen 5)

En la **(Imagen 6)** se muestra la visualización de la aplicación al momento de cargar uno de los archivos de texto, donde se podrá seleccionar un tipo de búsqueda, pueda ser no informadas como por ejemplo la búsqueda por amplitud o profundidad o en su defecto informadas como por ejemplo la búsqueda avara o estrella, posteriormente se calculara la cantidad de nodos creados, la profundidad del árbol de búsqueda, el costo de búsqueda y el tiempo de ejecución de la búsqueda seleccionada, para finalizar se podrá hacer clic en el botón iniciar búsqueda en donde se mostrara la simulación correspondiente a dicha búsqueda:



(Imagen 6)

Antes de proceder a explicar las diversas implementaciones de los algoritmos de búsqueda, es necesario entender el concepto bajo el cual se procede a crear los correspondientes nodos adyacentes al personaje, en la **(Imagen 7)** se visualiza el incremente y el decremento que se debe asignar para ir construyendo paso a paso el debido árbol de búsqueda:

	(X - 1 , Y)	
(X , Y - 1)	**(X , Y)**	**(X , Y + 1)**
	(X + 1 , Y)	

(Imagen 7)

IV. BUSQUEDA PREFERENTE POR AMPLITUD

La búsqueda preferente por amplitud es una estrategia sencilla en la que se expande primero el nodo raíz, a continuación se expanden todos los sucesores del nodo raíz, después sus sucesores, etc. Dicha estructura puede implementarse considerando la lista de nodos a expandir como una cola:

```
def searchPath(self):
    queue=deque([(self.startPosition[0],self.startPosition[1],None)])
    while(len(queue)>0):
        nodeList=queue.popleft()
        x=nodeList[0]
        y=nodeList[1]
        if(self.maze[x][y]=="M"):
            self.reconstructMaze()
            return(nodeList)
        if(self.maze[x][y]!="0"):
            if((self.maze[x][y]=="S")or(self.maze[x][y]=="C")):
                pass
            else:
                continue
        self.maze[x][y]="explored"
        for i in ([x,y-1],[x,y+1],[x-1,y],[x+1,y]):
            self.createdNodes+=1
            queue.append((i[0],i[1],nodeList))
    return([])
```

(Imagen 8)

V. BUSQUEDA DE COSTO UNIFORME

Se expande el nodo raíz, a cada nodo n del árbol se calcula el costo de ruta $g(n)$ y se expande el nodo de menor costo. Dicha estructura se puede implementar considerando la lista de nodos a expandir como una cola de prioridad, donde la prioridad es el costo y se selecciona aquel con menor prioridad:

```
def searchPath(self):
    queue=PriorityQueue()
    queue.put([0,self.startPosition[0],self.startPosition[1],None])
    while(queue.qsize()>0):
        nodeList=queue.get()
        x=nodeList[1]
        y=nodeList[2]
        if(self.maze[x][y]=="M"):
            self.reconstructMaze()
            return(nodeList)
        if(self.maze[x][y]!="0"):
            if((self.maze[x][y]=="S")or(self.maze[x][y]=="C")):
                pass
            else:
                continue
        self.maze[x][y]="explored"
        for i in ([x,y-1],[x,y+1],[x-1,y],[x+1,y]):
            self.createdNodes+=1
            addCosts=0
            for index in range(len(self.costPosition)):
                if([i[0],i[1]]==[self.costPosition[index][0],
                                self.costPosition[index][1]]):
                    addCosts+=self.costValue
            queue.put((self.calculateCost(nodeList,addCosts),
                       i[0],i[1],nodeList))
    return([])
```

(Imagen 9)

VI. BUSQUEDA PREFERENTE POR PROFUNDIDAD

La búsqueda preferente por profundidad explora cada camino posible hasta su conclusión (meta) antes de intentar otro camino. Esta técnica de búsqueda pertenece a las estrategias de búsqueda no informada, es decir la búsqueda no utiliza más que la información proporcionada por la definición del problema. Dicha estructura puede implementarse considerando la lista de nodos a expandir como una pila:

```
def searchPath(self):
    stack=[]
    stack.append([self.startPosition[0],self.startPosition[1],None])
    while(len(stack)>0):
        nodeList=stack.pop()
        x=nodeList[0]
        y=nodeList[1]
        if(self.maze[x][y]=="M"):
            self.reconstructMaze()
            return(nodeList)
        if(self.maze[x][y]!="0"):
            if((self.maze[x][y]=="S")or(self.maze[x][y]=="C")):
                pass
            else:
                continue
        self.maze[x][y]="explored"
        for i in ([x,y-1],[x,y+1],[x-1,y],[x+1,y]):
            self.createdNodes+=1
            stack.append((i[0],i[1],nodeList))
    return([])
```

(Imagen 10)

VII. BUSQUEDA AVARA

La búsqueda avara trata de expandir el nodo más cercano al objetivo a través de la implementación de la función heurística, alegando que probablemente conduzca rápidamente a una solución. Esta técnica de búsqueda pertenece a las estrategias de búsqueda informada, ya que utilizan conocimiento específico del problema más allá de la definición del problema en sí mismo, por lo que puede encontrar soluciones de manera más eficiente que una estrategia de búsqueda no informada. La búsqueda avara se puede implementar considerando la lista de nodos a expandir como una cola de prioridad, donde la prioridad es el valor de la heurística. Debido a la manera como se planteó la solución del problema la heurística para solucionar el mismo está basado en la formula geométrica para calcular la distancia entre dos puntos del plano cartesiano:

$$h(n) = Menor\ distancia\ en\ linea\ recta$$

Donde la distancia se calcula mediante la fórmula:

$$distancia = \sqrt{dx^2 + dy^2}$$

```
def searchPath(self):
    queue=PriorityQueue()
    queue.put([0,self.startPosition[0],self.startPosition[1],None])
    while(queue.qsize()>0):
        nodeList=queue.get()
        x=nodeList[1]
        y=nodeList[2]
        if(self.maze[x][y]=="M"):
            self.reconstructMaze()
            return(nodeList)
        if(self.maze[x][y]!="0"):
            if((self.maze[x][y]=="S")or(self.maze[x][y]=="C")):
                pass
            else:
                continue
        self.maze[x][y]="explored"
        for i in ([x,y-1],[x,y+1],[x-1,y],[x+1,y]):
            self.createdNodes+=1
            queue.put((self.calculateHeuristic(i[0],i[1]),
                       i[0],i[1],nodeList))
    return([])
```

(Imagen 11)

VIII. BUSQUEDA ESTRELLA

La búsqueda estrella se puede implementar considerando la lista de nodos a expandir como una cola de prioridad, donde la prioridad es el valor de $f(n)$ y se selecciona aquel con menor prioridad:

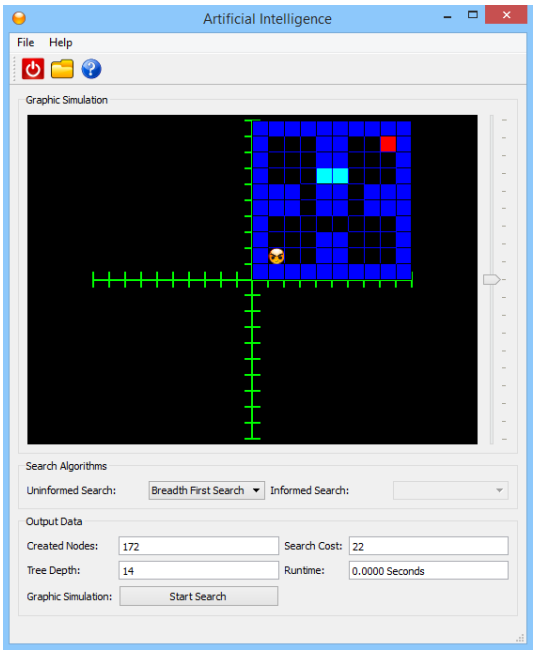
$$f(n) = g(n) + h(n)$$

Donde $g(n)$ nos da el coste del camino desde el nodo inicio al nodo n, y la $h(n)$ es el coste estimado del camino más barato desde n al objetivo, por tanto $f(n)$ representa el coste más barato estimado de la solución a través de n.

```
def searchPath(self):
    queue=PriorityQueue()
    queue.put([0,self.startPosition[0],self.startPosition[1],None])
    while(queue.qsize()>0):
        nodeList=queue.get()
        x=nodeList[1]
        y=nodeList[2]
        if(self.maze[x][y]=="M"):
            self.reconstructMaze()
            return(nodeList)
        if(self.maze[x][y]!="0"):
            if((self.maze[x][y]=="S")or(self.maze[x][y]=="C")):
                pass
            else:
                continue
        self.maze[x][y]="explored"
        for i in ([x,y-1],[x,y+1],[x-1,y],[x+1,y]):
            self.createdNodes+=1
            addCosts=0
            for index in range(len(self.costPosition)):
                if([i[0],i[1]]==[self.costPosition[index][0],
                                 self.costPosition[index][1]]):
                    addCosts+=self.costValue
            queue.put((self.calculateCost(nodeList,addCosts)+
                       self.calculateHeuristic(i[0],i[1]),
                       i[0],i[1],nodeList))
    return([])
```

(Imagen 12)

IX. PRUEBA – BUSQUEDA PREFERENTE POR AMPLITUD

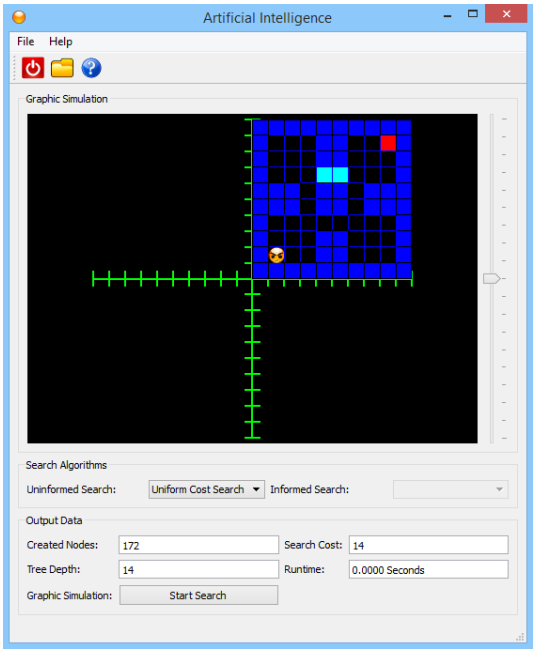


(Imagen 13)

```
(1,1),(1,2),(1,3),(2,3),(3,3),(3,4),(3,5),
(3,6),(4,6),(5,6),(6,6),(6,7),(6,8),(7,8),
(8,8)
```

(Imagen 14)

IX. PRUEBA – BUSQUEDA DE COSTO UNIFORME

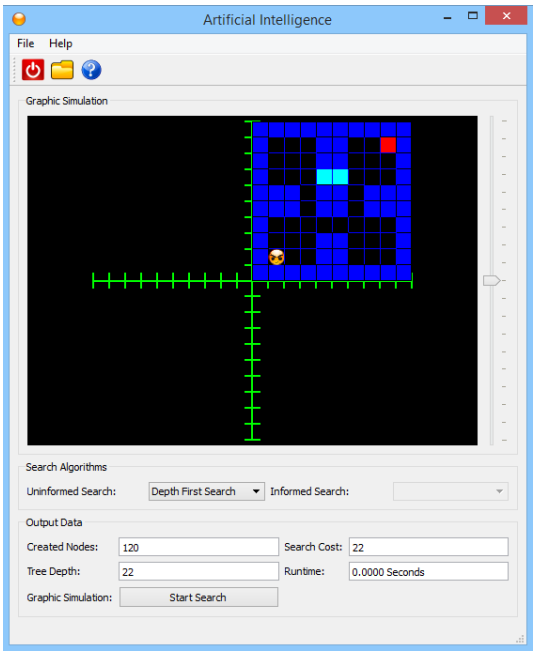


(Imagen 15)

```
(1,1),(1,2),(1,3),(2,3),(3,3),(4,3),(5,3),
(6,3),(6,4),(6,5),(6,6),(6,7),(6,8),(7,8),
(8,8)
```

(Imagen 16)

IX. PRUEBA – BUSQUEDA PREFERENTE POR PROFUNDIDAD

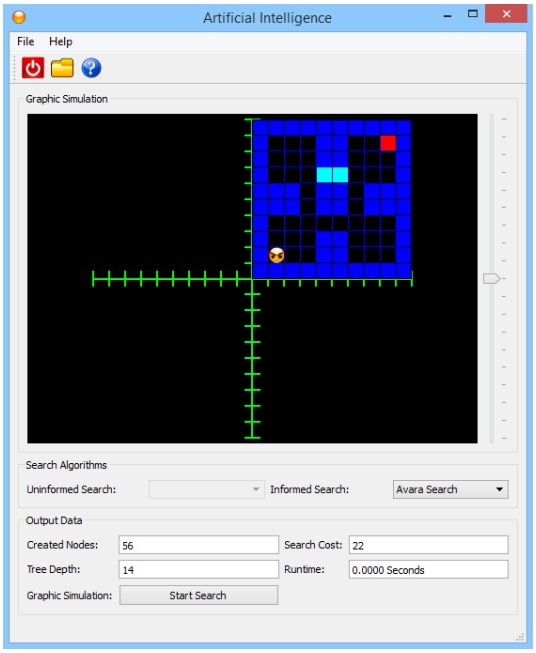


(Imagen 17)

```
(1,1),(2,1),(3,1),(3,2),(2,2),(1,2),(1,3),
(2,3),(3,3),(4,3),(5,3),(6,3),(6,4),(6,5),
(6,6),(7,6),(8,6),(8,7),(7,7),(6,7),(6,8),
(7,8),(8,8)
```

(Imagen 18)

IX. PRUEBA – BUSQUEDA AVARA

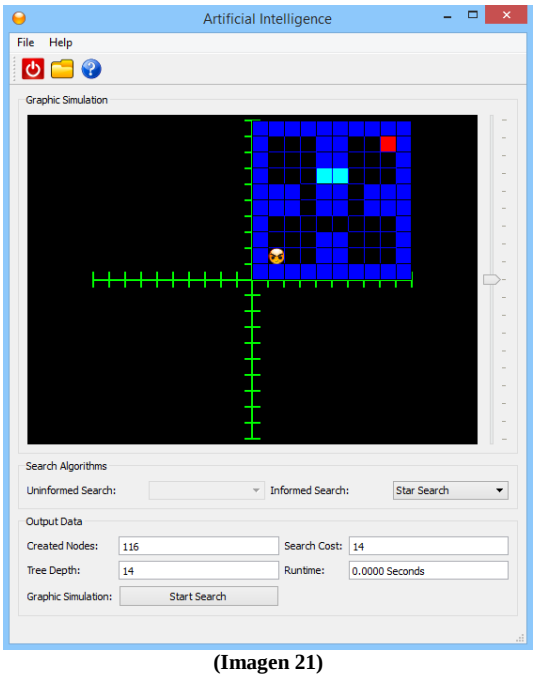


(Imagen 19)

```
(1,1),(1,2),(2,2),(2,3),(3,3),(3,4),(3,5),
(3,6),(4,6),(5,6),(6,6),(6,7),(7,7),(7,8),
(8,8)
```

(Imagen 12)

IX. PRUEBA – BUSQUEDA ESTRELLA



(Imagen 21)

```
(1,1),(1,2),(2,2),(2,3),(3,3),(4,3),(5,3),
(6,3),(6,4),(6,5),(6,6),(6,7),(7,7),(7,8),
(8,8)
```

(Imagen 22)

X. CONCLUSION

Las estrategias de búsquedas vistas en este trabajo nos dan una idea de cómo los investigadores en inteligencia artificial proponen diferentes formas de solución para los problemas. Estas técnicas son clásicas de la IA y es por ello que deben ser conocidas por todos aquellos que están relacionados con programación de soluciones por computadora. Existen otros métodos que requieren de mayor complejidad de programación para encontrar mejores soluciones en un tiempo razonable, como lo son el método de ascenso de la colina, el algoritmo de recocido simulado, los algoritmos genéticos y las redes neuronales. Todos ellos requieren de una mayor complejidad de computación y mayor conocimiento e información del problema.

En la **(Imagen 23)** se muestra una comparativa entre las estrategias de búsqueda descritas en el presente trabajo, esta nos puede dar una idea rápida de cuál estrategia debemos utilizar al intentar resolver un problema de búsqueda en espacio de estados.

BUSQUEDA	VENTAJAS	DESVENTAJAS
AMPLITUD	Encuentra la solución si existe dentro del espacio de búsqueda; fácil programación.	Requiere de mucha memoria para almacenar los nodos.
PROFUNDIDAD	Requisitos modestos de memoria; fácil implementación.	Puede tener ciclos infinitos y no encontrar el resultado.
AVARA	Puede encontrar buenas soluciones; resultados más eficientes que una búsqueda no informada.	Puede tener ciclos infinitos; requiere diseño de una heurística; complejidad de programación de la heurística.
ESTRELLA	Encuentra buenas soluciones; No se crean ciclos requerimientos moderados de memoria.	Requiere de muy buenas heurísticas.

(Imagen 23)

REFERENCIAS

[1] Python Algorithms - Mastering Basic Algorithms in the Python Language. Magnus Lie Hetland.

[2] Data Structures and Algorithms in Python. Michael T. Goodrich. Roberto Tamassia. Michael H. Goldwasser.

[3] Python para todos. Raúl González Duque.

[4] Inteligencia Artificial – Solución de Problemas por Búsqueda. Daniel Alejandro García.