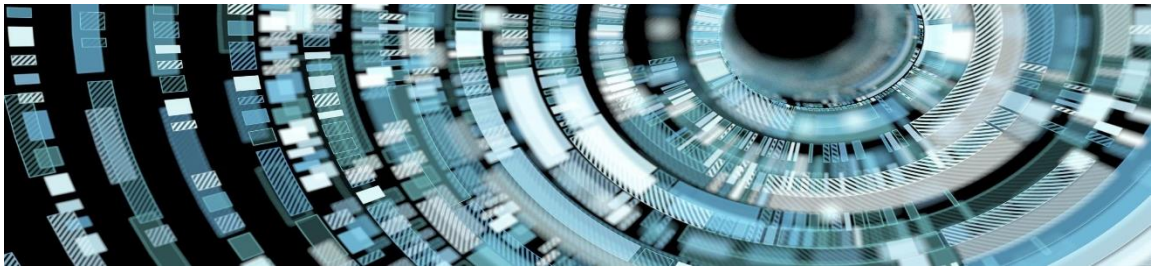

Informe Algoritmo A*

Alves Julio 66870 - Noguera Augusto 66865 - Ruiz Carolina 66711

Introducción

El algoritmo utilizado es el denominado A * (A estrella) que fue creado por primera vez por Hart El Al. (1968;1972) [1], el mismo utiliza elementos como los nodos (n), costos $g(n)$ entre conexiones de nodos, heurística $h(n)$ de las conexiones entre nodos (una estimación al nodo objetivo) y la función $f(n)$, como la suma entre $g(n) + h(n)$, para determinar el camino de menor costo desde un nodo inicial a un nodo final (objetivo), es decir:

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

Donde:

$g(n)$ Es el costo de llegar de llegar del nodo n-1 hasta el nodo n.

$h(n)$ Es la distancia aproximada (heurística) de llegar del nodo n hasta el nodo destino.

$f(n)$ Es el menor costo estimado de llegar del nodo n-a al nodo n.

Para que este algoritmo pueda funcionar la *heurística* debe cumplir con la propiedad de ser *admisible*, esto quiere decir, que no debe sobreestimar el costo real de llegar al nodo final. En el caso de este algoritmo utilizar una heurística admisible es tomar la distancia en línea recto de cada uno de los nodos al nodo objetivo donde se entiende que este tipo de distancia será la menor de todas por ello no se considera una heurística con sobrestimación.

También debemos tener en cuenta el concepto de condición de consistencia de una heurística donde si para cada nodo (n) y cada sucesor $'(n)$ de n generado por cualquier acción a , el coste estimado de alcanzar el objetivo desde (n) no es mayor que el coste de alcanzar $'(n)$ más el coste estimado de alcanzar el objetivo desde $'(n)$ [1]

EL concepto de Poda establece que para aquellos caminos poco prometedores se procede a "podarlos" o dejarlos de tener en consideración ya que el algoritmo al evaluar su falta de optimalidad establece no seguir por este camino.

Se debe entender que este tipo de algoritmos entonces solo funciona si contamos con una buena heurística, además de que debemos tener en cuenta que el nivel computacional y uso de recursos a medida que crece el número de nodos se establece en forma exponencial.

Modelo Implementado

Para iniciar la descripción de la implementación establecida, primero del lenguaje de programación elegido para desarrollar la aplicación es *Python.* Por consiguiente, se establece la estructura general de aplicación:

Se identifica al nodo inicial y al nodo final, si determinamos que el nodo inicial no es igual al nodo final ingresado se crean las siguientes listas: *nodos abiertos, nodos cerrados, nodos vecinos,* generar variable *nodo actual,* e insertar *nodo inicial* en lista de *nodos cerrados.*

Mientras al *nodo final* no lo encontremos en *nodos cerrados* el *nodo actual toma* el valor del último establecido en *nodos cerrados*.

Se llama al método *añadir vecino* y se cargan a la lista de *nodos vecinos* todos aquellos vecinos del nodo actual, menos aquellos que estén en la lista de *nodos cerrados,* esto quiere decir que ya fueron explorados. Para estos vecinos de la lista se calcula su valor función $f = g + h$ y se además se guarda su nodo padre (*nodo actual*).

Para cada nodo cargado de la lista de *nodos vecinos,* si el nodo evaluado no está en la lista de *nodos abiertos* ni en los *nodos cerrados* se debe establecer una copia del nodo evaluado en la lista de *nodos abiertos.* En el caso de que el nodo evaluado se encuentre en la lista de *nodos abiertos* y si el costo *g* del nodo evaluado es menor que el costo *g* del nodo de la lista de *nodos abiertos* entonces se procede a actualizar el costo *g* del nodo de la lista de *nodos abiertos* con los datos del nodo que se está evaluando.

Una vez realizado el recorrido de los nodos vecinos se debe vaciar dicha lista, se toma el nodo de la lista de *nodos abiertos* con menor *f* y se mueve a la lista de *nodos cerrados.* Este procedimiento se repite hasta encontrar el *nodo final*.

Una vez que se encontró en nodo final, la ruta optima se encuentra obteniendo el ultimo nodo de la lista de cerrados (el nodo solución, si existiere) y se van obteniendo los predecesores de cada nodo hasta llegar al nodo inicial que no tiene predecesor.

Los árboles y grafos que permiten realizar los gráficos se van generando a medida que se recorre el algoritmo. En él se van estableciendo las relaciones a través de la clase Tree, la cual a través de la biblioteca Graphviz va almacenando esas relaciones entre nodos y se van graficando las imágenes en el momento que se requiera.

Al iniciar la ejecución del algoritmo se generan todos los pasos del árbol de expansión por medio de una imagen para cada paso, las cuales son almacenadas en el disco para ser visualizados por la interfaz gráfica. En el caso de que se recorra dos veces el mismo nodo en el árbol de expansión (ya que puede existir otro camino de menor coste) el nodo que se genera lo hace indicando un numero delante del nombre del nodo, indicando la cantidad de veces que aparece el nodo en el árbol de expansión.

Modelo arquitectónico MVC propuesto

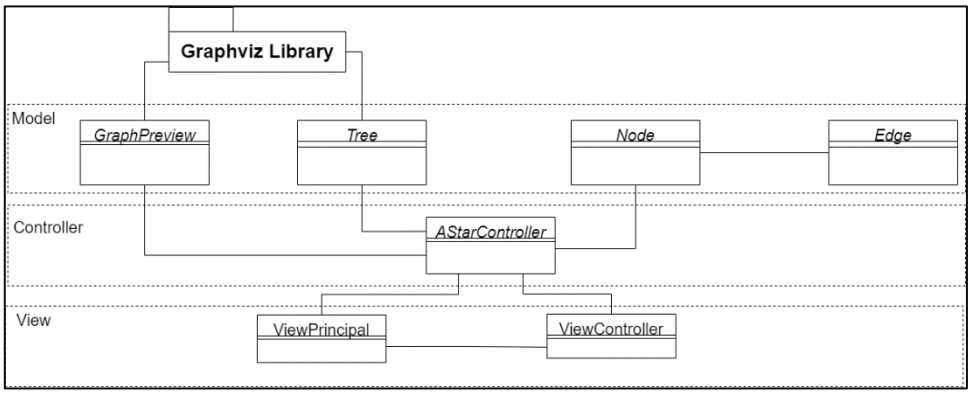

Figura 1. Modelo Arquitectónico MVC propuesto

El modelo arquitectónico de la Figura 1 representa la estructura interna de la solución propuesta. Se abstrae de clases que no formen parte del cuerpo del programa.

La utilización de dicha arquitectura permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos. [2]

AStarController: Representa al controlador principal del programa. Este se encarga de mantener encapsulada toda la lógica necesaria para la implementacion del algoritmo.

ViewController: Representa al controlador de la vista principal. En el se encuentra la logica de control de vistas.

ViewPrincipal: Representa a la ventana principal que contiene la implementacion grafica de los pasos necesarios (divididos por ventanas graficas) para recibir las entradas del usuario y comunicar al controlador los datos de la implementación del algoritmo, ya sea de manera manual o aleatoria.

Node: Es el modelo que una vez instanciado contiene información correspondiente a cada nodo. Como ser su nombre, costo (g), menor costo estimado (f), heurística (h) y su predecesor. Además, contiene información respecto a las aristas que se encuentra asociado el nodo, es decir, con que otros nodos se encuentra relacionado.

Edge: Es el modelo que representa las relaciones que tienen cada nodo, es decir, el camino de llegar de llegar de un nodo a otro, incluyendo el costo de ese camino.

Tree: Es el modelo de datos que contiene información respecto a los árboles de expansión de cada resolución. En ella, se mantiene los datos de colores de nodos, directorio de guardado de las imágenes generadas en la expansión y será la encargada de generar cada uno de los árboles de la solución de manera gráfica, a través de la biblioteca grafica “Graphviz”.

GraphPreview: Es el modelo de datos que contiene la misma información y lógica que el modelo de datos mencionado anteriormente “Tree” a diferencia que este se encarga de graficar los grafos de vista previa para las rutas cargadas o generadas de manera aleatoria.

Graphviz Library: Esta biblioteca permite realizar los gráficos de árboles y grafos implementados en la solución. Esto lo hace a través definiendo un lenguaje descriptivo DOT. En la solución se generan dichos archivos DOT acompañado de una imagen generada en formato PNG. Esto se almacena en el disco para poder ser observados en la vista.

Estructuras de Datos utilizadas

Clases

Se definieron dos clases *Node* y *Edge* para nodos y aristas respectivamente. Definimos además dentro de la clase *Node* una lista de nodos donde se almacenarán los nodos que se relacionan con el mismo, a modo de mejorar el funcionamiento.

Listas

edges = [] (aristas)

nodes = [] (nodos)

neighbors = [] (vecinos)

open_nodes = [] (nodos abiertos)

close_nodes = [] (cerrados)

short_path = [] (camino más corto)

Variables

end_node (nodo solucion)

start_node (nodo de comienzo)

current_node (nodo actual)

Lista de Librerías

Random (Utilizada para generación de números aleatorios) [3]

Numpy (Utilizada para distribución uniformes de numero aleatorios) [4]

PySimpleGUI (Utilizada para el desarrollo de la interfaz gráfica) [5]

Operator (Utilizado el método attrgetter para obtener el objeto en una lista) [6]

Copy (Utilizada para realizar copias superficiales de los objetos del tipo Nodo) [7]

Tiping (Utilizado para forzar el tipado en el retorno de funciones) [8]

Os (Método del SO utilizada para obtener rutas absolutas de las imágenes) [9]

Pruebas

Se establece una serie de pruebas en las cuales vamos a tener en cuenta tres casos particulares, tomaremos un grafo de 7 nodos con 9 aristas, primero teniendo una heurística considerablemente bien establecida:

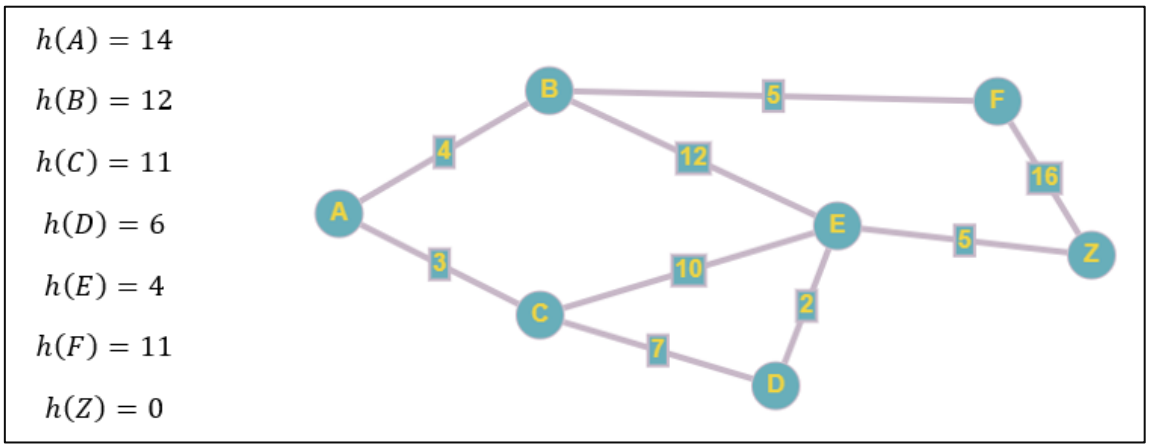

Figura 2. Grafo de prueba primer caso

Una vez corrido en el programa, desarrollado por el equipo, se puede obtener el siguiente resultado, donde vemos el recorrido que realiza el algoritmo, el concepto de poda implementado para caminos no prometedores como el A – B – F – Z y el marcado con color naranja para aquellos nodos abiertos explorados. Con ello llega al camino optimo Z.

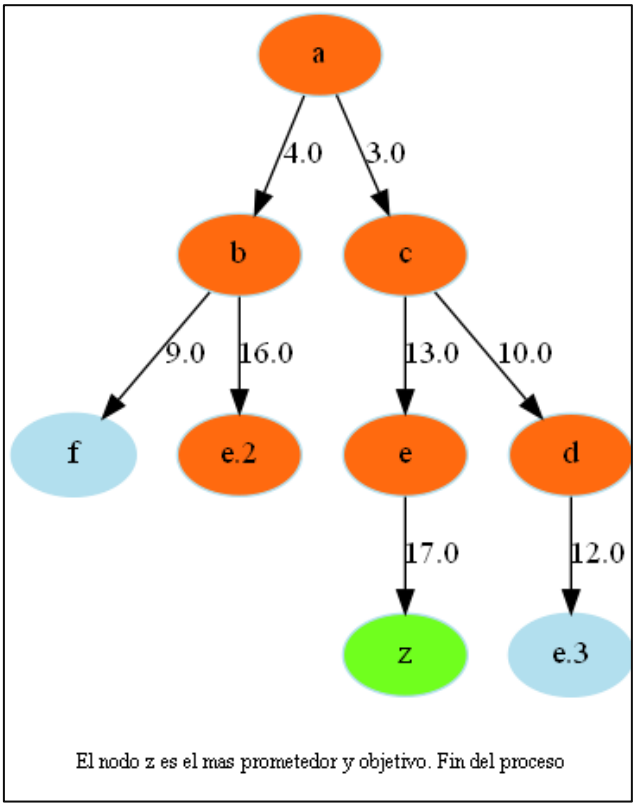

Figura 3. Árbol prueba primer caso

En un segundo caso y teniendo en cuenta el mismo grafo, pero con $h(n) = 1$ y costos $g(n) = 1$ obtenemos el siguiente resultado (diferente a cuando utilizamos heurísticas mejor establecidas), el algoritmo toma el camino A – B – F – Z

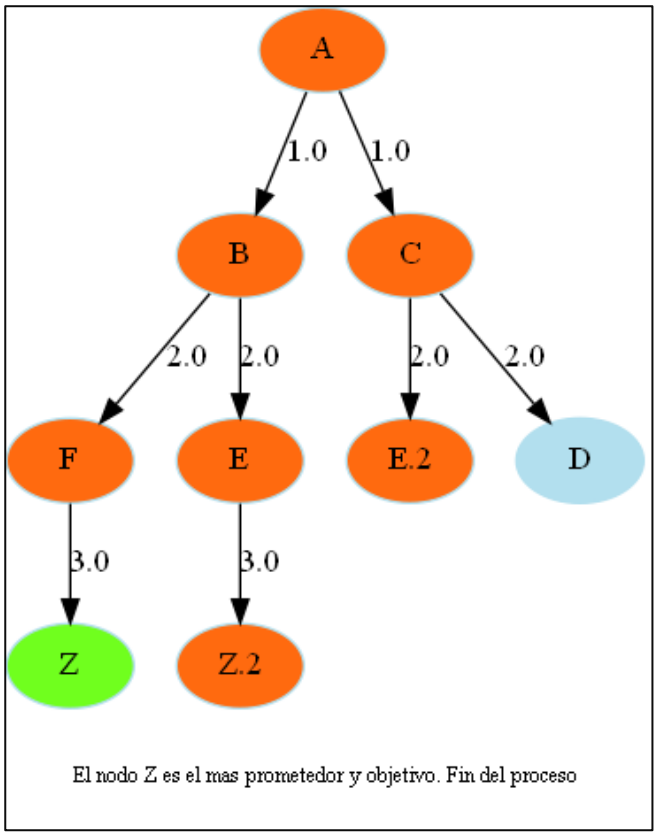

Figura 4. Árbol prueba segundo caso

En una tercera prueba se establece una sobreestimación sobre la heurística con respecto a los costos reales de los caminos de A hacia Z. A continuación, se establece un detalle de heurísticas sobrestimadas.

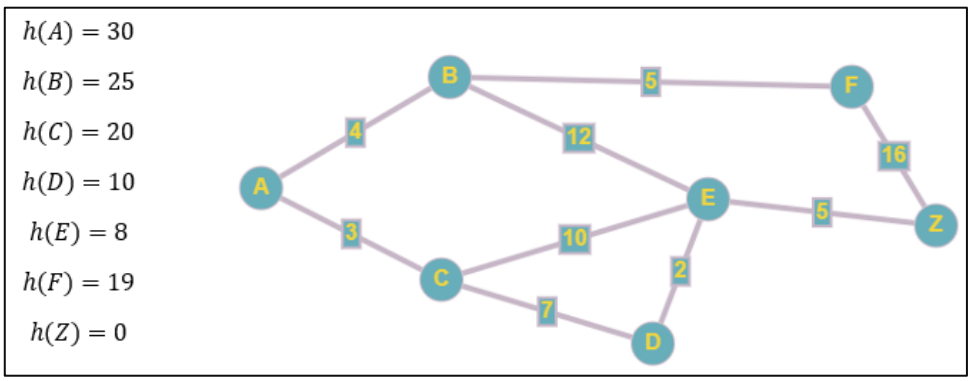

Figura 5. Grafo de prueba tercer caso

Se puede evidenciar en la siguiente imagen que por la sobrestimación encontramos que el algoritmo no explora la rama B.

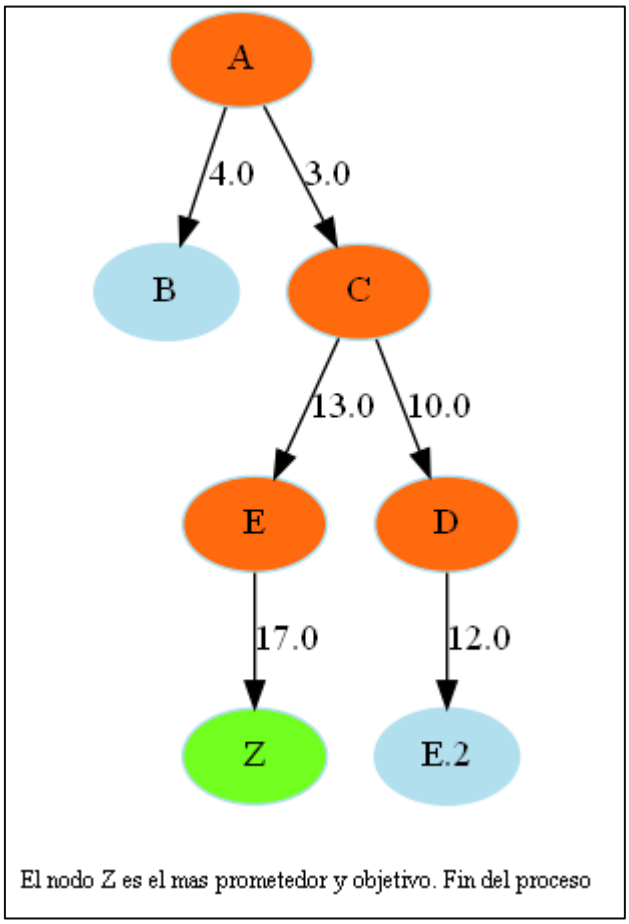

Figura 6. Árbol prueba tercer caso

Conclusiones:

A partir de la solución propuesta se observa claramente a través de las pruebas realizadas que el algoritmo implementado encuentra el camino de menor coste entre un nodo inicial y un nodo final, siempre que se cumplan las condiciones de admisibilidad mencionadas.

Se debe tener en cuenta que si los costos y las heurísticas son constantes el modelo implementado se convierte en un algoritmo de primero en profundidad, ya que se pierde el conocimiento que se tiene sobre los nodos. A su vez si se sobrestima la heurística el algoritmo puede que no encuentra la ruta optima.

Fuente Bibliográficas

[1] Russell, S. J. y Norvig, P. (2008). Inteligencia artificial: un enfoque moderno (2a. ed.). Pearson Educación. Capítulo 4 - Sección 4.1

[2] Sommerville (2011) Ingeniería de Software (9na ed.). Capitulo 6 – Sección 6.2

[3] https://docs.python.org/es/3.10/library/random.html

[4] https://numpy.org/doc/stable/

[5] https://pypi.org/project/PySimpleGUI/

[6] https://docs.python.org/es/3/library/operator.html

[7] https://docs.python.org/es/3/library/copy.html

[8] https://docs.python.org/es/3/library/typing.html

[9] https://docs.python.org/es/3.10/library/os.html