

Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0217 Estructuras abstractas de datos y algoritmos para ingeniería
II ciclo 2016

Proyecto 2

Algoritmo de búsqueda A*

Laura Rincón Riveros, B55863
Esteban Vargas Vargas, B16998
Grupo 1
Profesor: Ricardo Román

2 de diciembre de 2016

Índice

1. Reseña del algoritmo/estructura	4
2. Funcionamiento del algoritmo/estructura	6
3. Objetivos	11
3.1. Objetivo General	11
3.2. Objetivos específicos	11
4. Metodología	12
4.1. Implementación del algoritmo de búsqueda A* en C++	12
4.1.1. Lógica y funcionamiento de la búsqueda A*	12
4.1.2. Estructura del algoritmo	12
4.2. Representación gráfica	14
4.3. Tiempos de ejecución y complejidad computacional	14
5. Resultados	15
5.1. Implementación del algoritmo de búsqueda A* en C++	15
5.2. Representación gráfica mediante uso de Python	17
5.3. Tiempos de ejecución y complejidad computacional	18
5.3.1. Teórico	18
5.3.2. Experimental	18
6. Análisis de tiempos de ejecución y complejidad computacional	20
7. Conclusiones	21
8. Anexos	23
8.1. Código fuente del algoritmo de búsqueda A*	23
8.1.1. Node.h	23
8.1.2. Node.cpp	24
8.1.3. Matrix.h	26
8.1.4. Matrix.cpp	26
8.1.5. Astar.h	29
8.1.6. Astar.cpp	30
8.1.7. main.cpp	33
8.2. Código fuente de creación de imágenes en Python	33

Índice de figuras

1.	Algoritmo de Dijkstra con obstáculos [1]	4
2.	Algoritmo Best-First-Search con obstáculos [1]	5
3.	Algoritmo A-Estrella con obstáculos [1]	5
4.	Ejemplo 1 Representación costos G, H y F [3]	6
5.	Ejemplo 1 Representación parentesco de nodos [3]	7
6.	Ejemplo 2 Primera iteración algoritmo A-Estrella [4]	8
7.	Ejemplo 2 Segunda iteración algoritmo A-Estrella [4]	9
8.	Ejemplo 2 Resultado final algoritmo A-Estrella [4]	10
9.	Impresión en consola del resultado del programa	15
10.	Impresión en un archivo de texto de la salida del programa	16
11.	Representación visual salida programa	17
12.	Gráfico de Función de Tiempo experimental	19

Índice de tablas

1.	Valor por unidad de tiempo de los instructivos del programa.	14
2.	Datos de tiempo experimental	18

1. Reseña del algoritmo/estructura

Los algoritmos de búsqueda se desarrollan para encontrar caminos entre dos puntos, que ahorren recursos. Estos recursos pueden ser combustible, energía, tiempo, etc; por lo que su aplicación se extiende en varios ámbitos como por ejemplo videojuegos o mapas que simulen ciudades reales.

El primer algoritmo de este tipo que tuvo gran impacto fue el algoritmo Dijkstra, el cuál fue propuesto por el científico de la computación holandés Esger Dijkstra, alrededor de 1956. Este algoritmo encuentra, casi en todos los casos, el camino más corto entre dos nodos (que se pueden representar como casillas de una cuadrícula); sin embargo, al no utilizar heurística, recorre todo el mapa sin discriminar la cercanía de los nodos al nodo final. Ver figura 1.

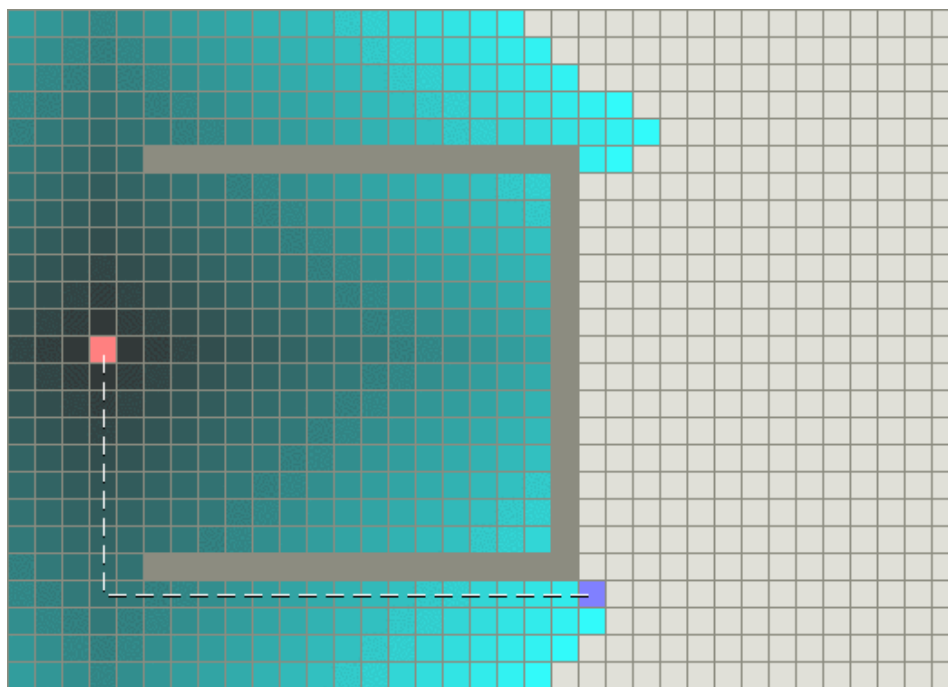


Figura 1: Algoritmo de Dijkstra con obstáculos [1]

Otro algoritmo de búsqueda conocido es el *Best-First-Search*, el cual trabaja de manera similar al algoritmo de Dijkstra con la diferencia de que toma en cuenta algo llamado heurística, que indica qué tan largo se encuentra cada nodo en el mapa del nodo final. Este algoritmo no garantiza encontrar el camino más corto entre dos puntos, ya que intenta moverse hacia el nodo final, aunque este camino no sea el más corto. Esto sucede ya que sólo considera el costo de llegar a la meta e ignora el costo del camino que va trazando.

Aún así, es más rápido que el algoritmo de Dijkstra ya que va analizando los nodos que se encuentran más cerca del nodo final, no los que están más cerca del inicial. Ver figura 2.

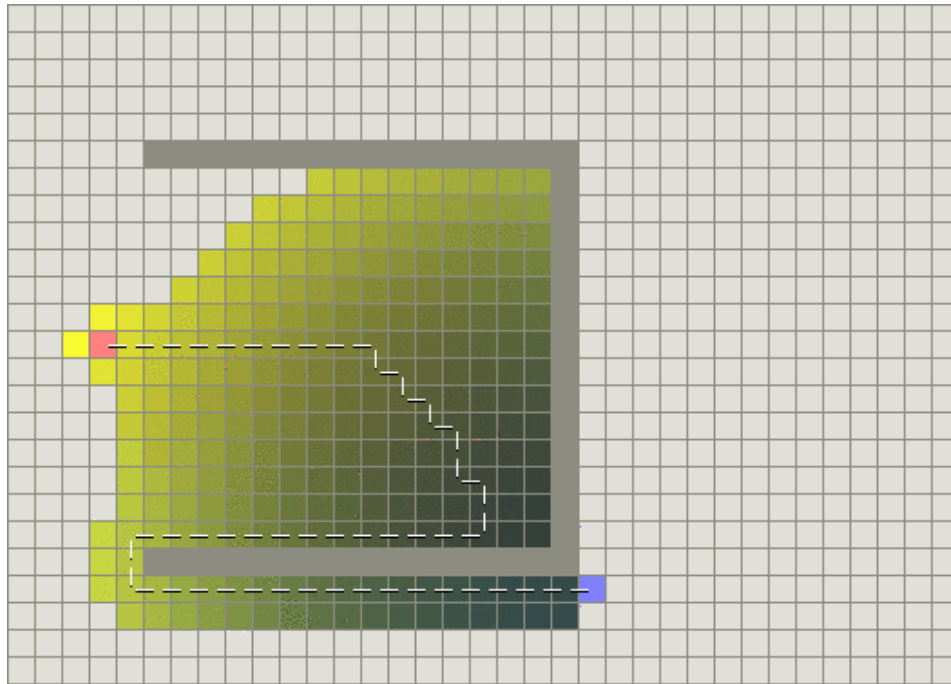


Figura 2: Algoritmo Best-First-Search con obstáculos [1]

El algoritmo A-Estrella fue propuesto en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael. Este algoritmo combina de cierta forma lo mejor de los dos algoritmos mencionados anteriormente. Por esto la gran característica del A-Estrella, que lo hace muy atractivo, es que a través de costos (G , H y F), favorece los nodos que se encuentran cerca del inicial así como los nodos que se encuentran cerca del final; encontrando así el camino más corto.

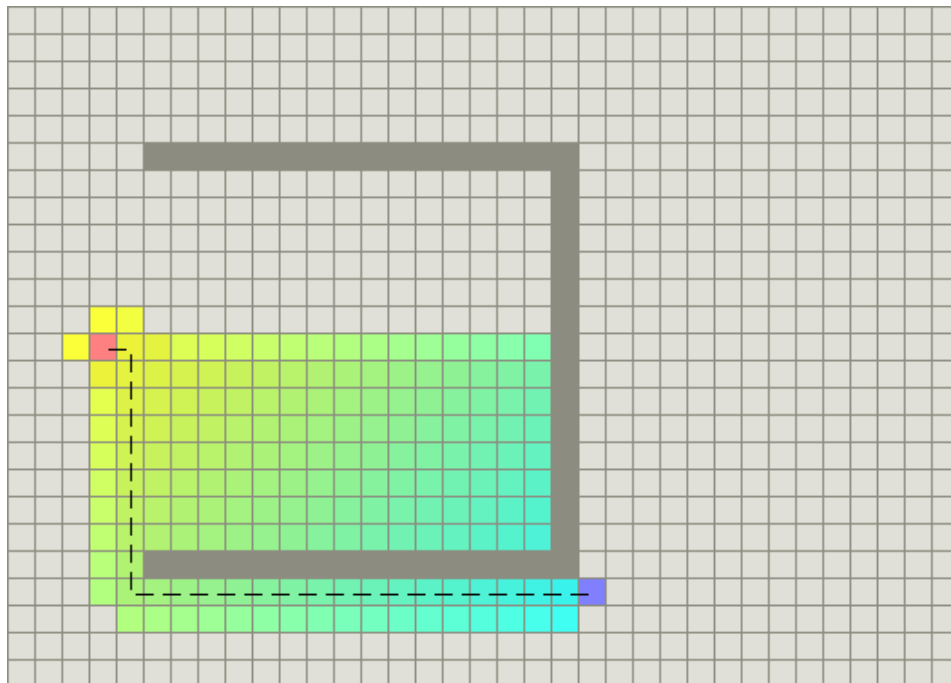


Figura 3: Algoritmo A-Estrella con obstáculos [1]

2. Funcionamiento del algoritmo/estructura

El algoritmo de búsqueda A-estrella busca el camino más corto entre dos puntos de un mapa, el cual se puede representar de distintas formas. En la presente implementación se utilizó una matriz de objetos *nodo*. Cada *nodo* cuenta con las siguientes características:

- Costo G: Establece qué tan lejos se encuentra un nodo cualquiera del nodo inicial. Representa el costo de ir desde un nodo cualquiera hasta el nodo inicial. El costo de movimiento entre dos nodos se puede calcular de distintas maneras, por ejemplo, en las implementaciones que se muestran como ejemplos, el costo de moverse entre dos nodos con movimiento diagonal es de 14; y con movimiento horizontal o vertical es de 10.
- Costo H: Representa la heurística de qué tan lejos se cree que se encuentra un nodo cualquiera del nodo final. Este valor se puede calcular igual cómo se indicó que se calcula el valor G (14 para movimiento diagonal y 10 para horizontal/vertical), o puede calcularse sólo tomando en cuenta desplazamientos horizontal y vertical.
- Costo F: Es simplemente la suma de los costos G y H.
- Nodo Padre: Es un puntero que indica hacia qué nodo debo moverme si se traza un camino que incluya a ese nodo.
- Vecindad: incluye los 8 nodos adyacentes a un nodo cualquiera, es decir los que colindan ya sea horizontalmente, verticalmente o de manera diagonal.

En la figura 4 se observa cómo para un nodo inicial A y un nodo final B, se calcularon los costos G (esquina superior izquierda), los costos H (esquina superior derecha) y los costos F (centro) de todos los nodos en la vecindad del inicial (pintados con verde). En la figura 5 se observa como los nodos verdes apuntan a su respectivo nodo padre.

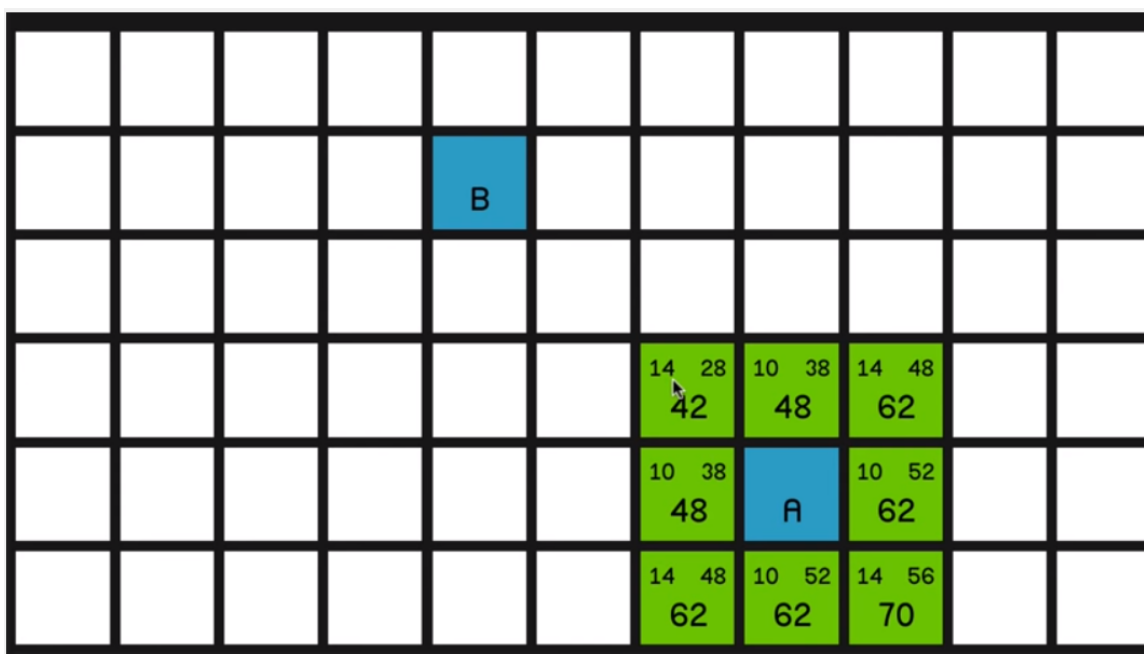


Figura 4: Ejemplo 1 Representación costos G, H y F [3]

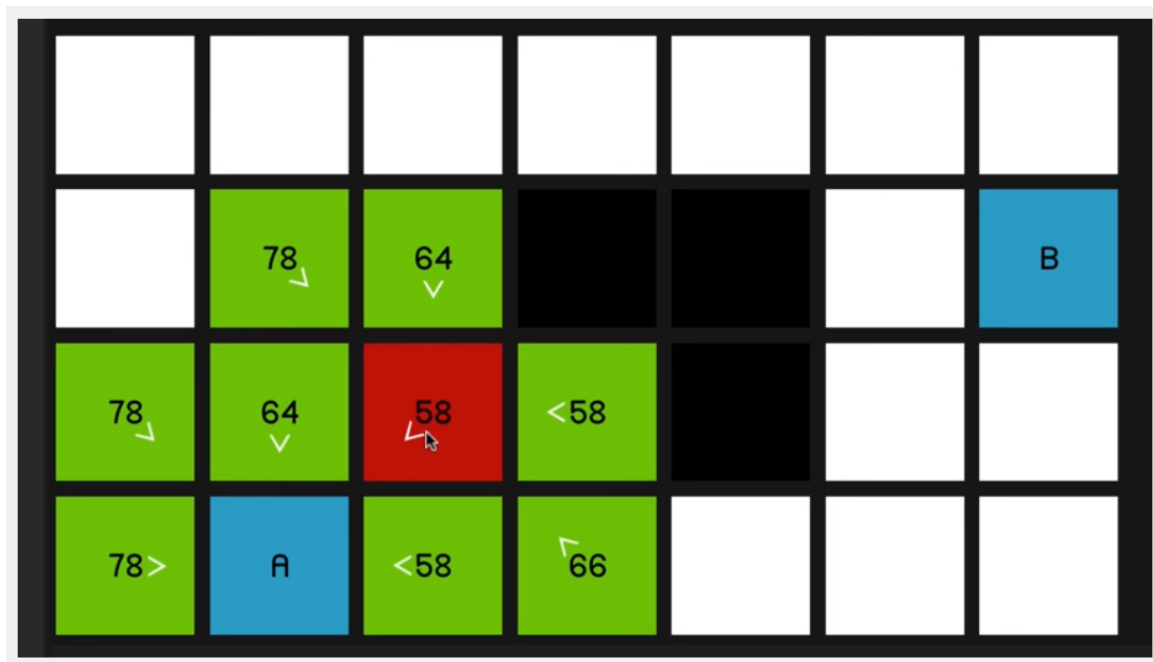


Figura 5: Ejemplo 1 Representación parentesco de nodos [3]

Para este algoritmo se usan dos listas: una *open list* y una *closed list*. La *open list* contiene los nodos a analizar y la *closed list* los ya analizados. Nótese de la figura 6 lo siguiente:

- Cada nodo se identifica por un número, el cual se encuentra en la esquina superior izquierda de cada celda.
- El punto azul indica el nodo inicial y el amarillo el nodo final.
- En la esquina superior derecha de cada nodo, representado por celdas, se encuentra el costo H. En este caso se utilizó la distancia de Manhattan, la cual toma en cuenta solo desplazamientos verticales y horizontales.
- El número que se encuentra de color celeste es el costo G. En este caso simplemente es 10 para todos los nodos que se encuentran horizontal o vertical del nodo inicial y 14 para los que se encuentran diagonal.
- El número de color verde es el costo F, que como se puede observar es la suma de los costos H y G.
- Las flechas de color amarillo indican que todos los nodos de la vecindad del nodo inicial tienen como padre el nodo mencionado.
- En la *open list* se encuentran todos los nodos de la vecindad del nodo inicial, los cuales no han sido analizados todavía. En la *closed list* está el único nodo que ha sido analizado, el nodo inicial.

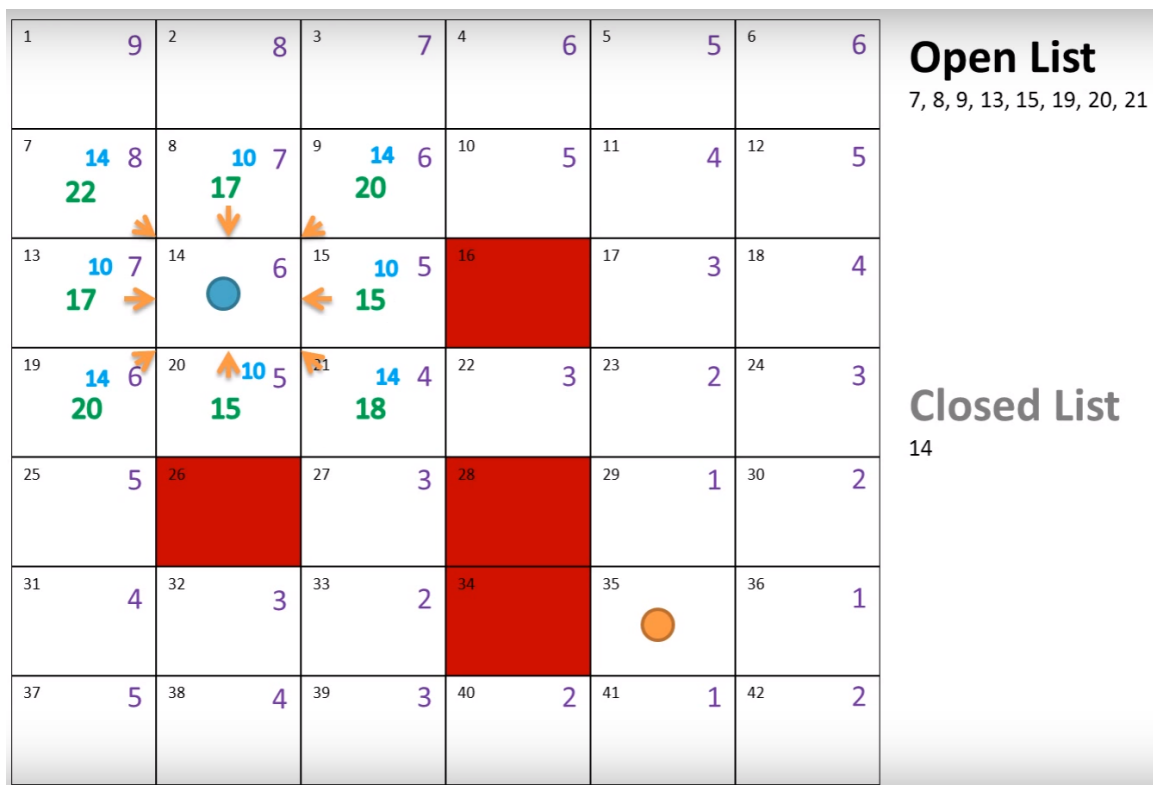


Figura 6: Ejemplo 2 Primera iteración algoritmo A-Estrella [4]

Se deben ir analizando los nodos hasta que se encuentre el camino más corto entre el final y el inicial. El siguiente nodo a analizar siempre es el nodo que se encuentre en la *open list* que tenga el costo F más bajo. Como se ve en la figura 7, el siguiente nodo a analizar fue el nodo 15, ya que su costo F era el menor. Al analizar el nodo 15, este debe eliminarse de la *open list* y añadirse a la *closed list*, como se observa al costado derecho de la figura 7.

Siempre que un nodo A , que se encuentre en la vecindad de un nodo B que se esté analizando, ya se encuentre en la *open list*, se debe verificar lo siguiente:

- Si el costo G de A es mayor que el costo G de A si tuviera como padre a B , entonces hacer a B el padre de A .
- En caso contrario no modificar el parentesco.

Se puede ver como en la figura 6, el nodo 8 se añadió a la *open list* por estar en la vecindad del nodo inicial. Además, a este nodo 8 se le asignó el nodo inicial como padre. Al analizar el nodo 15, se puede observar en la figura 6, que el nodo 8 también se encuentra en la vecindad y como ya estaba en la *open list* entonces se debió verificar la condición mencionada. Como el costo G de 8 (movimiento vertical = 10) no es mayor que el costo G de 8 si 15 fuera su padre (movimiento horizontal + movimiento diagonal = 24), entonces se deja el parentesco como está, es decir, 14 sigue siendo el padre de 8.

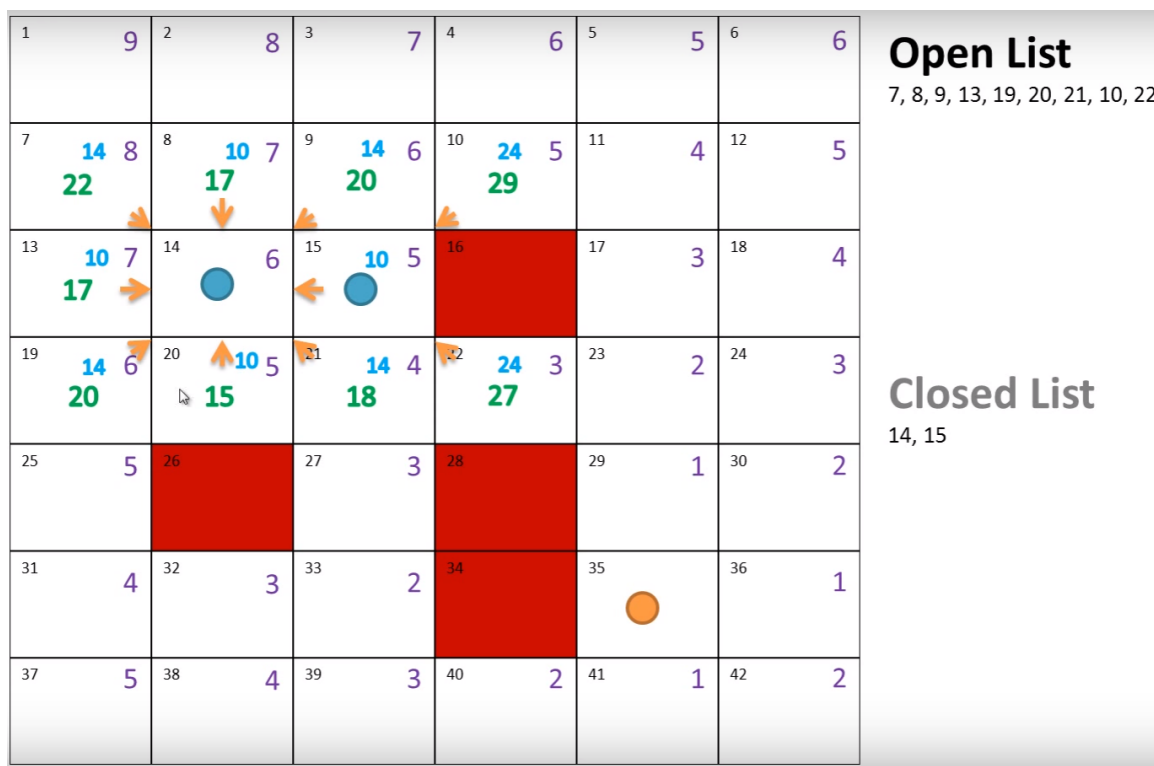


Figura 7: Ejemplo 2 Segunda iteración algoritmo A-Estrella [4]

Como se mencionó anteriormente, se van analizando los nodos que se encuentren en la *open list* que tengan el costo F más bajo. Esto se hace repetidas veces hasta llegar a un nodo C cuya vecindad contenga al nodo final. Cuando se da esto, se hace a C el padre del nodo final y se tiene ya el camino

más corto entre los nodos inicial y final. Este camino se encuentra siguiendo los parentezcos iniciando en el nodo final, hasta llegar al inicial. En la figura 8 se presenta el momento donde se encuentra el camino, el cual se representa con las flechas moradas. Como se puede observar en la figura, no fue necesario analizar todos los nodos del "mapa".

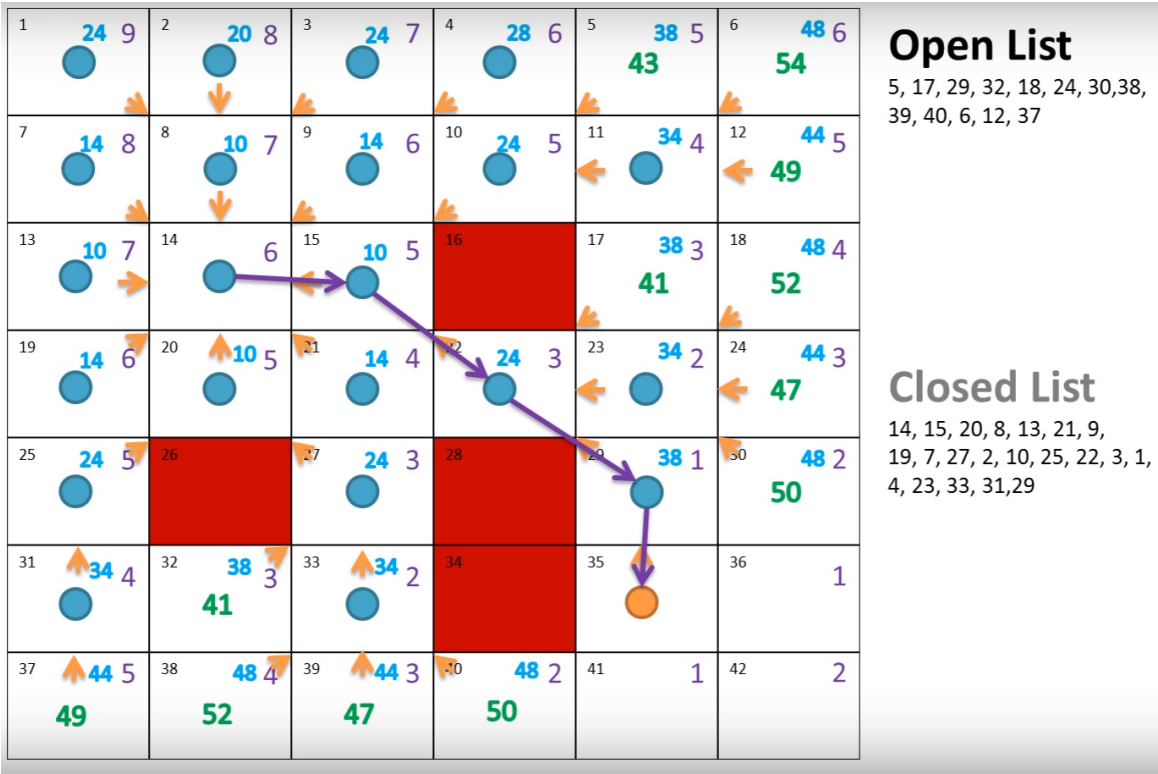


Figura 8: Ejemplo 2 Resultado final algoritmo A-Estrella [4]

3. Objetivos

3.1. Objetivo General

- Implementar el algoritmo de búsqueda A*

3.2. Objetivos específicos

- Encontrar y analizar la función de tiempo del algoritmo
- Obtener la complejidad del algoritmo implementado
- Comprobar el funcionamiento del programa ante distintas configuraciones de obstáculos

4. Metodología

En el desarrollo de la implementación del algoritmo de búsqueda A^* , se planteó el procedimiento que se presenta en las siguientes secciones con el fin de alcanzar los objetivos propuestos.

4.1. Implementación del algoritmo de búsqueda A^* en C++

4.1.1. Lógica y funcionamiento de la búsqueda A^*

Previo a la implementación del algoritmo, fue de gran importancia realizar una revisión bibliográfica sobre su funcionamiento, por lo que, se investigó sobre algoritmos de búsqueda similares al A^* .

Una vez hecha la investigación bibliográfica, ya manejando de manera teórica cómo funciona el algoritmo y qué es lo que busca, se procedió a trabajar en la estructura de este.

4.1.2. Estructura del algoritmo

Se desarrolló un algoritmo de búsqueda A^* en el lenguaje de programación C++. El programa contiene las siguientes funcionalidades:

- Capacidad de encontrar el camino más corto entre dos puntos con obstáculos.
- Reproducir, de manera visual, el camino entre dichos puntos.

A partir de las directrices anteriores, se procedió a crear la estructura del código del programa. Mediante la Programación Orientada a Objetos (POO), se utilizó la siguiente composición: ¹

- Clase Node: contiene las características descritas previamente de cada nodo.
 - Atributos:
 - + H_value: Valor heurístico de la distancia entre el nodo actual y el nodo final. Se utilizó la distancia de Manhattan, la cual sólo considera movimientos horizontales y verticales. Ambos con un costo de 1.
 - + G_value: Costo de movimiento entre el nodo actual y el nodo inicial. El movimiento horizontal entre dos nodos tiene un costo de 10 y el costo diagonal es de 14.
 - + F_value: Costo total, es decir, la suma de H_value y G_value.
 - + type: indentificador de tipo entero para diferenciar entre un nodo inicial (1), nodo final (2), nodo obstáculo (3), nodo parte del camino más corto entre el inicial y el final (4) o nodo común (0).
 - + parent: Puntero al nodo padre, para indicar el camino a seguir.
 - + pos_x: Columna en la que se encuentra el nodo dentro de la matriz.
 - + pos_y: Fila en la que se encuentra el nodo dentro de la matriz.
 - Funciones:
 - + Set_parent: Establece el padre de un nodo.

¹El signo + se refiere a los atributos y métodos públicos.

- + Set_start: Define el nodo inicial.
- + Set_goal: Define el nodo inicial.
- + Set_obstacle: Define a un nodo como obstáculo.
- + operator==
- + operator=
- + operator!=
- + operator<
- + operator>

Las ultimas funciones de sobrecarga de operadores se implementaron con el fin de poder comparar y asignar objetos de tipo Node.

- Clase Matrix: clase que contiene la matriz de nodos para representar el mapa.
 - Atributos:
 - + matrix: puntero doble de objetos Nodo.
 - + size: Dimensión de matriz cuadrada de nodos.
 - Funciones:
 - + set_G_value: Calcula los costos G para la vecindad del nodo inicial.
 - + get_G_value: Recalcula el G_value para un nodo en particular.
 - + Set_Movement_value: Función con el objetivo de comparar el G_value según el padre, para restablecer o no parentezco entre nodos.
 - + get_H_value: Establece el H_value para un nodo en específico.
 - + get_F_value: Establece el F_value para un nodo en específico.
 - + print_types
 - + print_G_values
 - + print_H_values
 - + print_F_values
 - + print_parents

Las funciones de print imprimen en consola, de manera matricial, los valores o coodenadas (parents) de los atributos requeridos.

- Clase Astar: Contiene la lógica del algoritmo.
 - Atributos:
 - + OpenList: vector que contiene los nodos pendientes de revisar.
 - + ClosedList: vector que contiene los nodos revisados.
 - + Mapa: objeto de tipo Matrix.
 - + current: nodo temporal utilizado para analizar la matriz de nodos.
 - Funciones:
 - + Set_Parent_G_Value: Establece el parentezco entre nodos.
 - + Add_To_Closed_List: Agrega un nodo a la ClosedList después de revisarlo.

- + Add_To_Open_List: Agrega un nodo a la OpenList para que sea revisado.
- + Find_path: Después de revisar todos los nodos, se utiliza Find_path para encontrar el camino más corto entre los puntos, mediante los padres de los nodos.
- + Remove_Node_Open_List: Elimina un nodo de la OpenList.
- + Get_Node_Open_List: Retorna el primero nodo en la OpenList, o sea, el nodo con el menor F_value.
- + Find_Node_Open_List: Regresa un valor booleano, con el fin, de confirmar si un nodo se encuentra en la OpenList.
- + Find_Node_Closed_List: Regresa un valor booleano, con el fin, de confirmar si un nodo se encuentra en la ClosedList.

4.2. Representación gráfica

El programa implementado en C++, imprime a manera de texto el contenido de la matriz de nodos. Se utilizó una biblioteca de Python capaz de recibir un archivo de texto con el contenido de la matriz, para luego crear una imagen con determinado número de pixeles.

Con ayuda de la biblioteca Pillow se construyó una imagen del mapa con el camino generado por el programa A*.

4.3. Tiempos de ejecución y complejidad computacional

Para obtener la función del tiempo de ejecución y la complejidad computacional fue necesario emplear la Tabla 1 para designar los valores en unidades de tiempo a los instructivos.

Tabla 1: Valor por unidad de tiempo de los instructivos del programa.

Valor en unidad de tiempo	Comando
1	Asignación e inicialización
1	Llamado a memoria
1	Operaciones de suma o resta
1	Comparación
1	Función cout
$1*c$	If(c), donde c es el número de comparaciones
$i*n$	for(), n son las veces que se repite e i son las intrucciones dentro del for
k^n	while(), repite n veces k instructivos

De esta manera, se analizaron las líneas de comando necesarias para la ejecución del programa y se calculó teóricamente la función de Tiempo.

Posteriormente, se utilizó la biblioteca Time.h para medir experimentalmente el tiempo que requería el algoritmo para su ejecución con el fin de comparar la función teórica con los resultados experimentales. Para las pruebas experimentales se utilizó una computadora Dell con sistema operativo Ubuntu 14.04, Intel(R) Core(TM) i3- 3277U CPU con 1.9GHz.

Con el debido análisis de la función de Tiempo, se procedió a establecer la complejidad del algoritmo implementado.

5. Resultados

A continuación se presentan los resultados obtenidos. Estos se presentan divididos en secciones acorde con la metodología, para facilitar una mejor comprensión.

5.1. Implementación del algoritmo de búsqueda A* en C++

² Con el objetivo de comprobar la funcionalidad del código, se ejecutó con varias configuraciones de obstáculos para asegurar que se estuviera encontrando el camino más corto entre los puntos inicial y final. A continuación se presenta un ejemplo, en la figura 9, con matrices de valores enteros³.

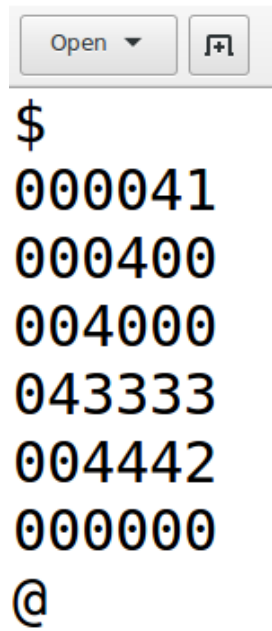
----- MAPA -----					
0	0	0	0	4	1
0	0	0	4	0	0
0	0	4	0	0	0
0	4	3	3	3	3
0	0	4	4	4	2
0	0	0	0	0	0

Figura 9: Impresión en consola del resultado del programa

El código se implementó de tal manera que la solución se imprimiera tanto en consola como en un archivo de texto out.txt; el cual sería posteriormente leído por el programa de Python para crear las imágenes del camino entre los puntos. El archivo de texto generado para la configuración de la figura 9 se muestra en la figura 10.

²Se puede disponer del Código fuente en la sección de Anexos.

³Se denota con un (1) para el nodo de inicio, (2) para el nodo final, (3) para los obstáculos y un (4) para el camino generado



\$
000041
000400
004000
043333
004442
000000
@

Figura 10: Impresión en un archivo de texto de la salida del programa

5.2. Representación gráfica mediante uso de Python

Para poder representar la salida del programa de una manera gráfica se procedió a crear un código en Python utilizando la biblioteca *Pillow*. A partir del archivo out.txt mostrado en la sección anterior, se creó una imagen pixel por pixel. Al tener una imagen se procedió a hacerle una modificación de tamaño. En la figura 11 se muestra la salida para la configuración de la figura 9 donde los puntos inicial y final se pintaron de azul, los obstáculos de rojo y el camino encontrado por el algoritmo de verde.

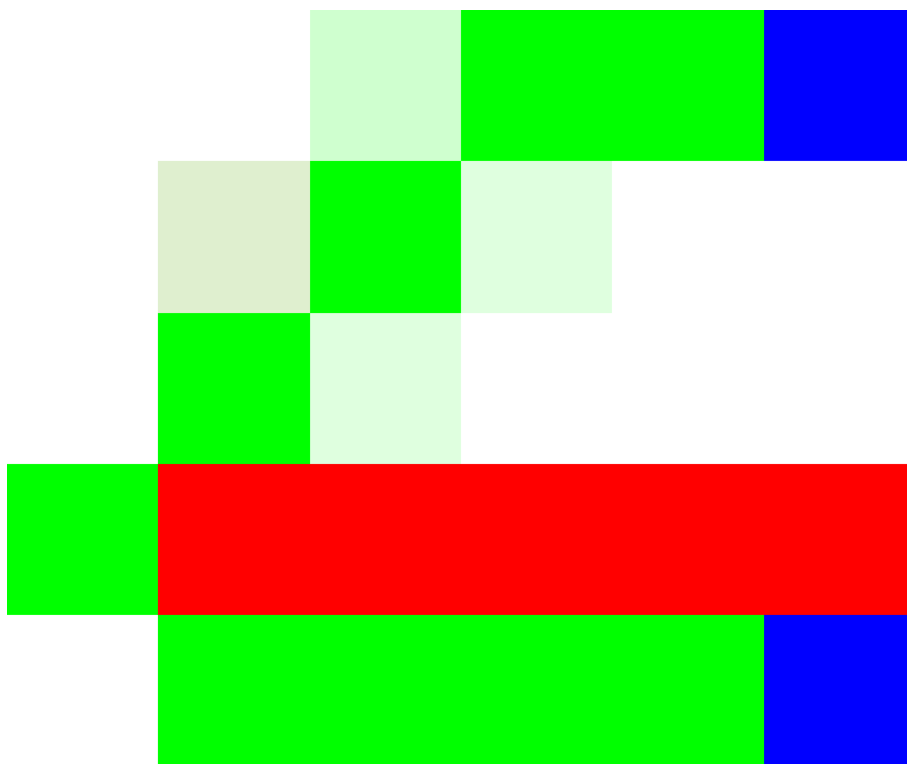


Figura 11: Representación visual salida programa

5.3. Tiempos de ejecución y complejidad computacional

5.3.1. Teórico

Se utilizó la Tabla 1 para modelar la función de tiempo, con lo que se obtuvo la expresión dada por la ecuación (1).

$$T(n) = (8 + k)^n + \frac{n}{2} \quad (1)$$

En la cual, n se refiere al número de nodos en la matriz y k es un constante que resume los instructivos utilizados. Además, el 8 se refiere al factor de ramificación en la búsqueda del punto final, es decir, este se refiere a los 8 nodos vecinos que se revisan para cada nodo analizado.

El primer factor de la expresión matemática (1) proviene de la búsqueda del punto final desde el nodo inicial (se requirió de *While*). El segundo término, resulta del *traceback*, es decir, de recorrer los nodos (de padre en padre) desde el nodo final hasta el nodo inicial para establecer el camino encontrado por el algoritmo.

5.3.2. Experimental

Para verificar experimentalmente la función de Tiempo se utilizó la ecuación (1), se colocaron los puntos a unir en los extremos diagonales de la matriz y obstáculos entre ellos, la prueba se realizó ampliando la matriz de nodos.

Los resultados del tiempo para dicho patrón se presentan en la tabla 2. Como complemento se muestra la gráfica respectiva de la función de tiempo experimental en la figura 12.

Tabla 2: Datos de tiempo experimental

Número de nodos (n)	Tiempo experimental (ut)								
	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5	Prueba 6	Prueba 7	Prueba 8	Promedio
64	0.00066	0.00094	0.000812	0.00933	0.000865	0.000864	0.00066	0.000685	0.00185
256	0.00991	0.008251	0.009499	0.001049	0.009603	0.008198	0.008417	0.010564	0.00819
1024	0.133495	0.133092	0.12768	0.167653	0.133474	0.157097	0.135519	0.13727	0.14066
4096	0.173611	0.130087	0.13235	0.128968	0.131108	0.136127	0.127905	0.125536	0.13571
16384	27.7417	27.7581	27.9996	27.8231	27.9801	27.9599	27.7649	27.7898	27.8522
65536	405	447	468	398	433	449	424	432	432

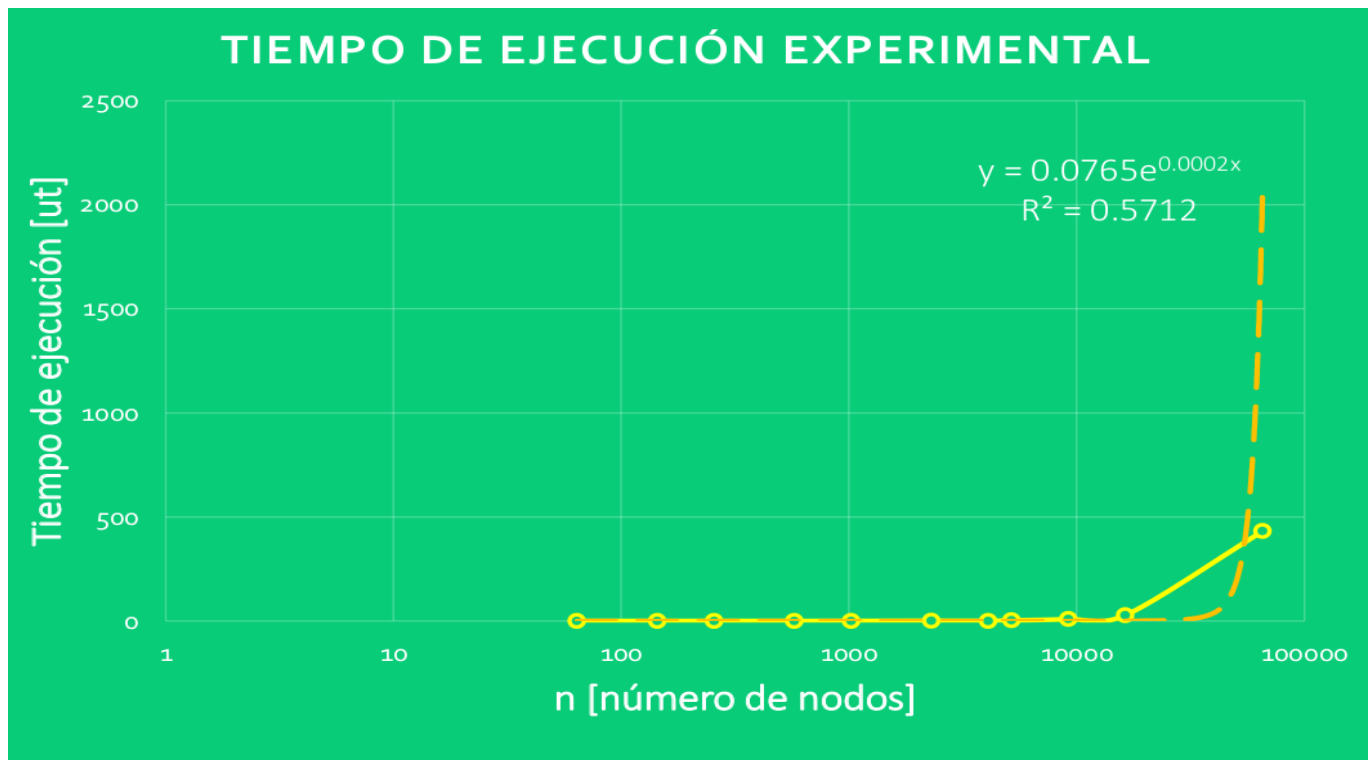


Figura 12: Gráfico de Función de Tiempo experimental

Los resultados obtenidos con la función de tiempo, muestran que el programa posee una complejidad computacional dada por la ecuación (2).

$$\Theta(n) = (8 + k)^n \quad (2)$$

6. Análisis de tiempos de ejecución y complejidad computacional

Con base en los resultados de la prueba experimental, dados en la Tabla 1, se puede rescatar la forma característica de la función. En la Figura 12 se puede notar una función exponencial, con lo cuál, se puede analizar que la función teórica acota superiormente a la función experimental y alcanza la asíntota en un n inferior, es decir, el tiempo de ejecución va a tender a infinito para un valor de n muy grande.

Dado que la gráfica de tiempo experimental es consistente con su forma exponencial, se concluye que el programa posee una complejidad computacional dada por la ecuación (2), que también se puede deducir de la función de Tiempo de la ecuación (1), ya que el segundo término es despreciable ante el valor de la exponencial.

En esta función de complejidad se describe el comportamiento de algoritmo en el peor de los casos donde se tiene que analizar cada uno de los nodos en la matriz, para llegar al punto final. Sin embargo, una de las ventajas de la búsqueda en A^* es que esos casos son limitados y por lo mismo, se considera una mejora al algoritmo de Dijkstra y el BFS, aún cuando ellos posean complejidades logarítmicas.

7. Conclusiones

- Se implementó de manera satisfactoria el algoritmo de búsqueda A^* en C++.
- Se comprobó el buen funcionamiento del algoritmo mediante la representación visual de algunas configuraciones.
- Se encontró la función de tiempo teórica y se comprobó mediante experimentos.
- La complejidad del algoritmo de interés es exponencial, con un factor de ramificación de ocho.

Referencias

- [1] Patel, A. (2016). *Heuristics: From Amit's Thoughts on Pathfinding*.
URL: <http://theory.stanford.edu/~amitp/GameProgramming/>
(Obtenido: 11/14/16).
- [2] Lester,P. (2005). *A* Pathfinding for Beginners*.
URL: <http://www.policyalmanac.org/games/aStarTutorial.htm>
(Obtenido: 11/14/16).
- [3] *A* Pathfinding (E01: algorithm explanation)*.
URL: <https://www.youtube.com/watch?v=-L-WgKMFuhE>
(Obtenido: 11/14/16).
- [4] *A* Pathfinding tutorial*.
URL: <https://www.youtube.com/watch?v=KNXfSOx4eEE>
(Obtenido: 11/14/16).
- [5] Reddy,H (2013). *PATH FINDING: Dijkstra's and A* Algorithm's*.
URL: <http://cs.indstate.edu/hgopireddy/algor.pdf>
(Obtenido: 11/14/16).

8. Anexos

8.1. Código fuente del algoritmo de búsqueda A*

A continuación se muestran los archivos creados para la implementación del algoritmo de búsqueda A*.

8.1.1. Node.h

```
#ifndef NODE_H
#define NODE_H

class Node {
public:

    int F_value;
    int G_value;
    int H_value;
    int type; /////0 para nodo normal, 1 para start, 2 para goal, 3 para obstaculo y 4 para path
    Node* parent;
    int pos_x;
    int pos_y;

    Node();

    Node(const Node &N);

    Node(int x, int y);

    Node(int f, int g, int h, int type, Node* father);

    virtual ~Node();

    void set_parent(Node p);

    void set_start();

    void set_goal();

    void set_obstacle();

    int operator==(const Node &N) const;

    Node& operator=(const Node &N);

    int operator!=(const Node &N) const;

    int operator<(const Node &N) const;

    int operator>(const Node &N) const;

private:
```

```
};
```

```
#endif /* NODE_H */
```

8.1.2. Node.cpp

```
#include "Node.h"
```

```
Node::Node() {  
    F_value = 0;  
    G_value = 0;  
    H_value = 0;  
    parent = 0x0;  
    type = 0;  
    pos_x=0;  
    pos_y=0;  
}
```

```
Node::Node(const Node &N) {  
    F_value = N.F_value;  
    G_value = N.G_value;  
    H_value = N.H_value;  
    parent = N.parent;  
    type = N.type;  
    pos_x=N.pos_x;  
    pos_y=N.pos_y;  
}
```

```
Node::Node(int x, int y) {  
    F_value = 0;  
    G_value = 0;  
    H_value = 0;  
    parent = 0x0;  
    type = 0;  
    pos_x=x;  
    pos_y=y;  
}
```

```
Node::Node(int f, int g, int h, int type, Node* father) {  
    F_value = f;  
    G_value = g;  
    H_value = h;  
    parent = father;  
    type = type;  
}
```

```
int Node::operator==(const Node &N) const {  
    if(this->pos_x==N.pos_x && this->pos_y==N.pos_y) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}
```



```

}

Node& Node::operator=(const Node &N) {
    this->F_value=N.F_value;
    this->G_value=N.G_value;
    this->H_value=N.H_value;
    this->type=N.type;
    this->pos_x=N.pos_x;
    this->pos_y=N.pos_y;
    this->parent=N.parent;
    return *this;
}

int Node::operator!=(const Node &N) const{
    if(this->pos_x!=N.pos_x || this->pos_y!=N.pos_y){
        return 1;
    }
    else{
        return 0;
    }
}

int Node::operator<(const Node &N) const{
    if(this->F_value < N.F_value){
        return 1;
    }
    else{
        return 0;
    }
}

int Node::operator>(const Node &N) const{
    if(this->F_value > N.F_value){
        return 1;
    }
    else{
        return 0;
    }
}

Node::~~Node() {

}

void Node::set_parent(Node p){
    this->parent=&p;
}

void Node::set_start() {
    this->type=1;
}

void Node::set_goal() {

```

```

        this->type=2;
    }

    void Node::set_obstacle() {
        this->type=3;
        this->G_value=1000000;
    }

```

8.1.3. Matrix.h

```

#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <string.h>
using namespace std;
#include "Node.h"
#include <cmath>           // std::abs

class Matrix {
public:

    Matrix(int tam);

    virtual ~Matrix();

    Node** matrix;

    int size;

    void set_G_value(Node &start);

    int get_G_value(int actual_G, int padre_G);

    int get_Movement_value(Node &actual, Node &Padre);

    int get_H_value(Node &actual, Node &final);

    int get_F_value(Node &actual);

    void print_types();
    void print_G_values();
    void print_H_values();
    void print_F_values();
    void print_parents();

};

#endif /* MATRIX_H */

```

8.1.4. Matrix.cpp

```

#include "Matrix.h"
#include <string.h>

```

```

#include <iostream>
using namespace std;

Matrix::Matrix(int tam){
    ///Creacion de la matriz
    matrix = new Node*[tam];
    size = tam;

    for (int i = 0; i < size; i++) {
        matrix[i] = new Node[size];
    }
    ///Inicializacion
    for(int row=0; row<size; row++){
        for (int column=0; column<size; column++){
            matrix[row][column]= Node(row, column);

        }
    }
}

//Matrix::Matrix(const Matrix &M){
//}

Matrix::~~Matrix() {
    delete matrix;
}

void Matrix::set_G_value(Node &start){
    int dx;
    int dy;
    for (int i=0; i<size; i++){
        for (int j=0; j<size; j++){
            if(matrix[i][j].type!=3){
                dx = abs(start.pos_x-i);
                dy = abs(start.pos_y-j);
                if (i==start.pos_y || j== start.pos_x){
                    if(dx>dy){
                        matrix[i][j].G_value= 14*dx+10*(dx-dy);
                    }
                    else{
                        matrix[i][j].G_value= 14*dy+10*(dy-dx);
                    }
                }
                else{
                    matrix[i][j].G_value= 10*(dx+dy);
                }
            }
        }
    }
}

int Matrix::get_G_value(int actual_G , int padre_G){

```

```

    int g = actual_G + padre_G;

    return g;
}

int Matrix::get_Movement_value(Node &a, Node &padre){
    int m=0;
    if(a.pos_x==padre.pos_x || a.pos_y==padre.pos_y){
        m=10;
    }
    else{
        m=14;
    }
    return m;
}

int Matrix::get_H_value(Node &actual, Node &final){
    int h = abs((final.pos_y-actual.pos_y))+abs((final.pos_x-actual.pos_x));
    return h;
}

int Matrix::get_F_value(Node &actual){
    int f = actual.G_value+actual.H_value;
    return f;
}

void Matrix::print_types(){
    ofstream myfile ("out.txt", ios::app);/**NUEVO**
    myfile<< "$"<<endl;/**NUEVO**
    cout<<"----- MAPA -----"<<endl;
    for(int i = 0; i < size; i++){
        for(int j=0; j< size ;j++){
            cout << matrix[i][j].type<< "\t";
            myfile << matrix[i][j].type;/**NUEVO**
        }
        cout << endl;
        cout << endl;
        myfile<< endl;/**NUEVO**
    }
    myfile<<"@"<<endl;
    //cout<<"-----"<<endl;
    myfile.close();/**NUEVO**
}

void Matrix::print_G_values(){
    cout<<"----- G_VALUES -----"<<endl;
    for(int i = 0; i < size; i++){
        for(int j=0; j< size ;j++){
            cout << matrix[i][j].type<< "\t";
        }
        cout << endl;
        cout << endl;
    }
}

```

```

    //cout<<"-----"<<endl;
}

void Matrix::print_H_values() {
    cout<<"----- H_VALUES -----"<<endl;
    for(int i = 0; i < size; i++){
        for(int j=0; j< size ;j++){
            cout << matrix[i][j].H_value<< "\t";
        }
        cout << endl;
        cout << endl;
    }
    //cout<<"-----"<<endl;
}

void Matrix::print_F_values() {
    cout<<"----- F_VALUES -----"<<endl;
    for(int i = 0; i < size; i++){
        for(int j=0; j< size ;j++){
            cout << matrix[i][j].F_value<< "\t";
        }
        cout << endl;
        cout << endl;
    }
    //cout<<"-----"<<endl;
}

void Matrix::print_parents() {
    cout<<"----- PADRES -----"<<endl;
    for(int i = 0; i < size; i++){
        for(int j=0; j< size ;j++){
            if(!matrix[i][j].parent){
                cout << "--" << "\t";
            }
            else{
                cout << matrix[i][j].parent->pos_x<< " " <<matrix[i][j].parent->pos_y<< "\t";
            }
        }
        cout << endl;
        cout << endl;
    }
    //cout<<"-----"<<endl;
}
}

```

8.1.5. Astar.h

```

#include "Matrix.h"
#include <vector>
#include <algorithm>
#ifndef ASTAR_H
#define ASTAR_H

class Astar{
public:
    vector<Node> OpenList;

```

```

vector<Node> ClosedList;
Matrix* Mapa;
Node current;

Astar();

Astar(int tam, int start_x, int start_y, int goal_x, int goal_y);

void Set_Parent_G_Value(Node actual);
void Add_To_Closed_List(Node actual);
void Add_To_Open_List(Node actual);

void Find_path(Node &current);

void Remove_Node_Open_List();

Node Get_Node_Open_List();

int Find_Node_Open_List(const Node a);

int Find_Node_Closed_List(const Node b);

virtual ~Astar();

};

#endif /* ASTAR_H */

```

8.1.6. Astar.cpp

```

#include "Astar.h"

Astar::Astar() {
    Mapa = new Matrix(20);
}

int Astar::Find_Node_Open_List(const Node a) {
    int b = 0;
    vector<Node>::iterator it;
    it = find(OpenList.begin(), OpenList.end(), a);
    if (it != OpenList.end()) {
        b = 1;
    } else {
        b = 0;
    }
    return b;
}

Node Astar::Get_Node_Open_List() {
    vector<Node>::iterator it;
    sort(OpenList.begin(), OpenList.end());
    it = OpenList.begin();
    return *it;
}

```

```

void Astar::Remove_Node_Open_List() {
    vector<Node>::iterator it;
    it = OpenList.begin();
    OpenList.erase(it);
}

void Astar::Find_path(Node &current) {

    if (current.type==0){
        current.type = 4;
    }
    if(current.parent){
        Find_path(*current.parent);
    }
}

int Astar::Find_Node_Closed_List(const Node a) {
    int b = 0;
    vector<Node>::iterator it;
    it = find(ClosedList.begin(), ClosedList.end(), a);
    if (it != ClosedList.end()) {
        b = 1;
    } else {
        b = 0;
    }
    return b;
}

void Astar::Add_To_Closed_List(Node actual) {
    ClosedList.push_back(actual);
}

void Astar::Add_To_Open_List(Node actual) {
    OpenList.push_back(actual);
}

void Astar::Set_Parent_G_Value(Node actual) {
    //Hace que el padre de todos estos nodos de la vecindad sea el inicial
    for (int i = actual.pos_x - 1; i <= actual.pos_x + 1; i++) {
        for (int j = actual.pos_y - 1; j <= actual.pos_y + 1; j++) {
            if (i >= 0 && i < Mapa->size && j >= 0 && j < Mapa->size) {
                if (i == actual.pos_x && j == actual.pos_y) {
                } else {
                    if (Find_Node_Closed_List(Mapa->matrix[i][j]) == 0
                        && Find_Node_Open_List(Mapa->matrix[i][j]) == 0) { //Punto de inflexion
                        Mapa->matrix[i][j].parent = &(Mapa->matrix[actual.pos_x][actual.pos_y]);

                        Mapa->matrix[i][j].G_value =
                            Mapa->get_Movement_value
                                (Mapa->matrix[actual.pos_x][actual.pos_y], Mapa->matrix[i][j]);
                    }
                }
            }
        }
    }
}

```

```

    }
}

Astar::Astar(int tam, int start_x, int start_y, int goal_x, int goal_y) {
    Mapa = new Matrix(tam);
    Mapa->matrix[start_x][start_y].set_start();
    Mapa->matrix[goal_x][goal_y].set_goal();
    Mapa->set_G_value(Mapa->matrix[start_x][start_y]);
    Mapa->matrix[4][1].set_obstacle();
    //Mapa->matrix[0][3].set_obstacle();
    Mapa->matrix[1][3].set_obstacle();
    Mapa->matrix[2][3].set_obstacle();
    //Mapa->matrix[3][3].set_obstacle();
    Mapa->matrix[4][3].set_obstacle();
    Mapa->matrix[5][3].set_obstacle();
    Mapa->matrix[6][3].set_obstacle();

    //Obtencion de la heuristica (H Value) de los nodos mediante distancia de Manhattan
    for (int i = 0; i < Mapa->size; i++) {
        for (int j = 0; j < Mapa->size; j++) {
            Mapa->matrix[i][j].H_value =
                Mapa->get_H_value(Mapa->matrix[i][j], Mapa->matrix[goal_x][goal_y]);
        }
    }

    //Agrega el nodo inicial a la closed list

    Add_To_Open_List(Mapa->matrix[start_x][start_y]);
    int confirm = 0;

    while(confirm!=1){
        confirm = Find_Node_Closed_List(Mapa->matrix[goal_x][goal_y]);
        current = Get_Node_Open_List();
        Add_To_Closed_List(current);
        Set_Parent_G_Value(current);

        if (!OpenList.empty()) {
            Remove_Node_Open_List();
        }
        for (int i = current.pos_x - 1; i <= current.pos_x + 1; i++) {
            for (int j = current.pos_y - 1; j <= current.pos_y + 1; j++) {
                if (i >= 0 && i < Mapa->size && j >= 0 && j < Mapa->size) {
                    Node temp = Mapa->matrix[i][j];

                    if(temp!=Mapa->matrix[start_x][start_y]){
                        if(temp.parent){
                        }
                        if (!(i == current.pos_x && j == current.pos_y)) {

                            if (temp.type == 3 || Find_Node_Closed_List(temp) == 1) {
                                }

```



```

        else{
            if (Mapa->get_Movement_value(current, temp)<
                Mapa->get_Movement_value(*(temp.parent), temp)
                || Find_Node_Open_List(temp) == 1) {

                temp.parent = &current;
                Mapa->matrix[i][j].G_value =
                Mapa->get_G_value(temp.G_value, temp.parent->G_value);
                temp.F_value = temp.G_value + temp.H_value;
            }
            if (Find_Node_Open_List(temp) == 0 && temp.type!=1) {
                Add_To_Open_List(temp);
            }
        }
    }
}

Find_path(current);
}

Astar::~~Astar() {
}

```

8.1.7. main.cpp

```

#include <cstdlib>
#include "Matrix.h"
#include "Astar.h"

using namespace std;

int main(int argc, char** argv) {

    Astar A = Astar(6,0,5,4,5); //tamano 6, 0 y 5 coordenadas nodo inicial
    //4 y 5 coordenadas nodo final
    A.Mapa->print_types();
    return 0;
}

```

8.2. Código fuente de creación de imágenes en Python

```

from PIL import Image, ImageDraw, ImageFont, ImageColor
import sys
#ejemplo de ejecucion: python b.py a.csv 500 500

#reader = open('~/.NetBeansProjects/A-Star/out.txt', "r")
maxRow=int(sys.argv[2])
maxCol=int(sys.argv[3])
with open(sys.argv[1], 'r') as f:

```

```
a=f.readline().strip()
b=0
c=0
while a != '@':
    if(a=='$'):
        c+=1
        out = Image.new("RGB", (int(sys.argv[2]),int(sys.argv[3])))#no tocar
        dout = ImageDraw.Draw(out)#salida que crea una imagen
        a=f.readline().strip()
        for i in range(0,maxRow):
            for j in range(0,maxCol):
                if(a[j]=="1" or a[j]=="2"):
                    dout.point((j,i),(0,0,255))#pixel 0 y 1 pintar de color 2
                elif(a[j]=="4"):
                    dout.point((j,i),(0,255,0))#pixel 0 y 1 pintar de color 2
                elif(a[j]=="3"):
                    dout.point((j,i),(255,0,0))#pixel 0 y 1 pintar de color 2
                else:
                    dout.point((j,i),(255,255,255))#pixel 0 y 1 pintar de color 2
        a=f.readline().strip()
        out.save("out"+ str(c)+".png" , "PNG")
```