FACULTAD DE INGENIERÍA

FORMATO
PRÁCTICAS DE LABORATORIO

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

NOMBRE DE LA ASIGNATURA: *INTELIGENCIA ARTIFICIAL*
NOMBRE DEL PROFESOR: *RAMIREZ ORTEGON JUSTINO*
NOMBRE DEL ALUMNO: *ALBA ARCOS CHRISTOPHER GREGORIO*
TEMA: *PROYECTO: ALGORITMO A**

OBJETIVO:
Crearemos un programa que nos permita la visualización de la ejecución de el algoritmo de búsqueda A* en una cuadricula, donde este reconocerá obstáculos, inicio y destino.

RESUMEN:
En una cuadricula de 12x12 se nos permite visualizar en donde se encuentra el destino, inicio y los obstáculos, al momento de iniciar el procesamiento se nos muestra la visualización del algoritmo, como calculo y determina la ruta. Además de las características requeridas para la tarea se han agregado ciertas funciones adicionales, como lo son la de generar los puntos antes dicho de forma aleatoria o que un laberinto sea generado. Además de que el tamaño de la cuadricula puede ser modificado y la visualización de todo esto es en tiempo real. A pesar de que se nos especifico que el algoritmo no debía permitir pasos en diagonales, los he agregado como una opción extra, de la misma forma para el cómo se determina la distancia he incluido las distancias Euclidiana y de Manhattan, por lo que obtendremos distintitos resultados que pueden surgir de las combinaciones de estos dos últimos parámetros.

MARCO TEÓRICO:
A* (estrella) es un algoritmo informático que fue presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael. Es un algoritmo muy utilizado para buscar una ruta posible y eficiente entre dos puntos. Esta ruta que ha de tener el menor coste posible se produce entre un origen y un destino llamados nodos. A* es un algoritmo heurístico, ya que hará uso de una función de evaluación heurística. Esta que es una de sus principales características se basa en que etiqueta los diferentes nodos de la red según el coste de llegar al objetivo. Esta forma de etiquetar los nodos está compuesta a su vez por otras dos funciones: 1) una muestra la distancia actual desde el nodo origen hasta el nodo a etiquetar. 2) la otra indica la distancia desde este nodo a etiquetar hasta el nodo destino. Esto le sirve, en función de los valores obtenidos, de establecer la probabilidad que tienen los nodos de la red de formar parte de la ruta más eficiente. Existen diferentes heurísticas, pero solo son admisibles si nunca sobreestiman el coste de alcanzar el objetivo. Es decir, solo son válidas cuando en el punto actual la estimación del coste de alcanzar el objetivo nunca es mayor que el menor coste posible.

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

El algoritmo A* solo puede utilizarse cuando se cumplen unas determinadas condiciones. Además, hay situaciones donde podría no funcionar bien por diferentes razones. Una desventaja práctica muy importante del algoritmo A* es su necesidad de almacenamiento ya que guarda en su memoria todos los nodos generados. Otra de sus limitaciones es cuando existen factores dinámicos, es decir, cuando el terreno es modificable o existen objetos móviles. Por ejemplo, algunos videojuegos que tienen mapas muy grandes son muy exigentes en cuando a requerimientos en tiempo real, así como en cuanto a almacenamiento de datos y tiempo de procesador. Aunque hubiera suficiente memoria para realizar los cálculos, podría ser que estos fueran ineficientes e inadecuados.

Este algoritmo utiliza una función de evaluación f(n) = g(n) + h'(n), donde h'(n) representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final, y g(n), el costo real del camino recorrido para llegar a dicho nodo, n. A* mantiene dos estructuras de datos auxiliares, que podemos denominar abiertos, implementado como una cola de prioridad ordenada por el valor f(n) de cada nodo, y cerrados, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados. El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad: mientras que h'(n) tiende a primero en profundidad, g(n) tiende a primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

Función heurística de A*:

- f (n) = g(n) + h(n): Coste real del plan (camino) de mínimo coste que pasa por n.
- f* (n) = g(n) + h*(n): estimación de f.

Estrategia de A*:

- Entre las hojas del árbol de búsqueda, elegir el nodo de valor f* mínimo.

Interpretación fuerte de A*:

- Una heurística suele facilitar la resolución de un problema, pero no garantiza que se resuelva.
- Una heurística es una regla de tres para un problema.
- Búsqueda: Optimalidad o incluso completitud no garantizados.

Esquematización de A*:

- Se basa en la búsqueda general.
- Almacenar el valor g de cada nodo expandido.
- Mantener la estructura abierta ordenada por valores crecientes de f*.
- Insertar nuevos nodos en la estructura abierta según sus valores de f*.

EQUIPO Y SOFTWARE UTILIZADO:

- Computadora personal (HP Omen con Core i7 y 8 Gb de RAM)
- El software de Processing
- Java

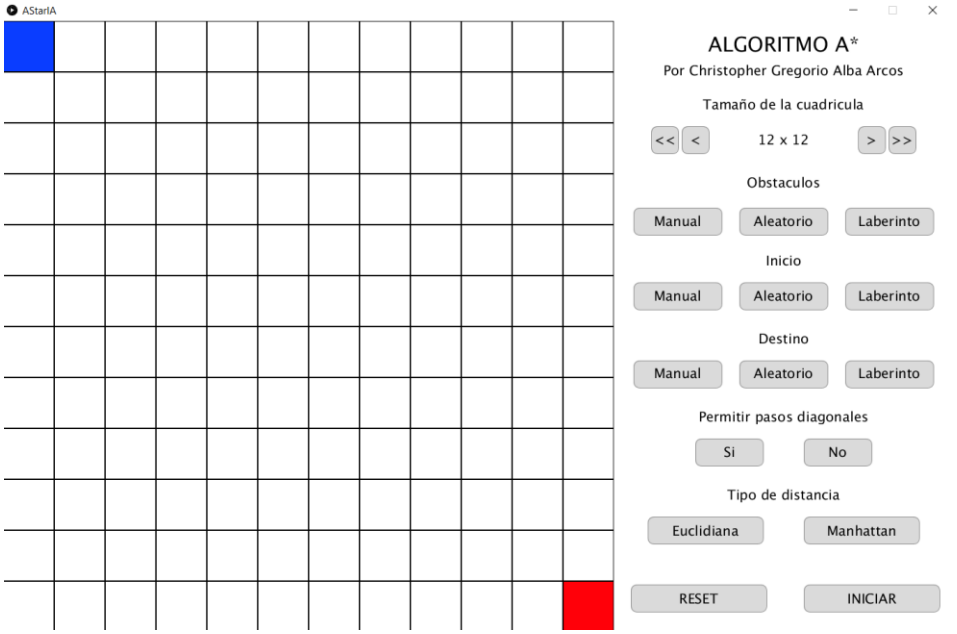
FACULTAD DE INGENIERÍA

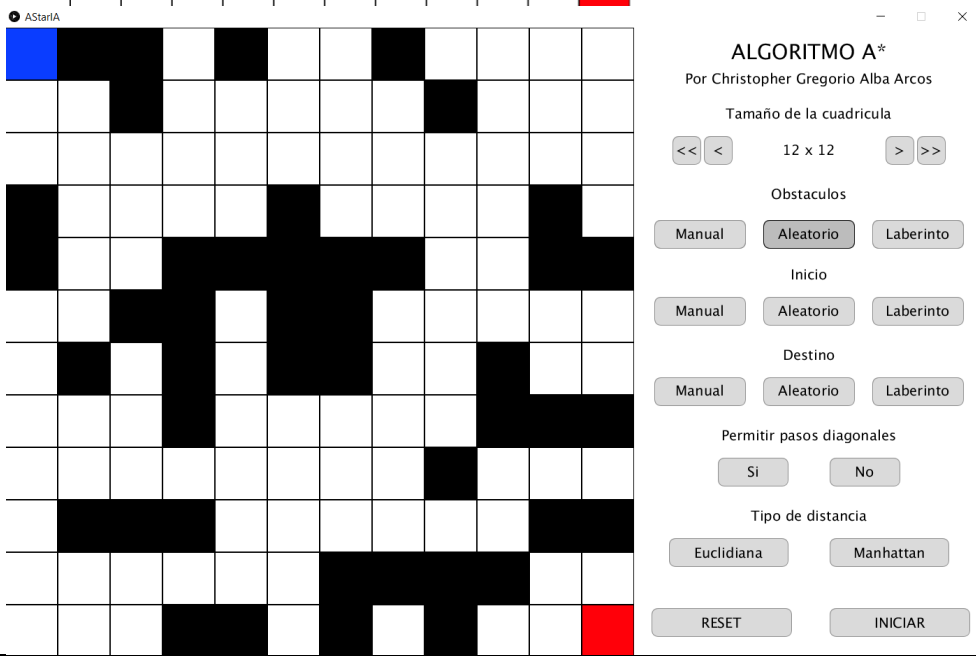
FORMATO
PRÁCTICAS DE LABORATORIO

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

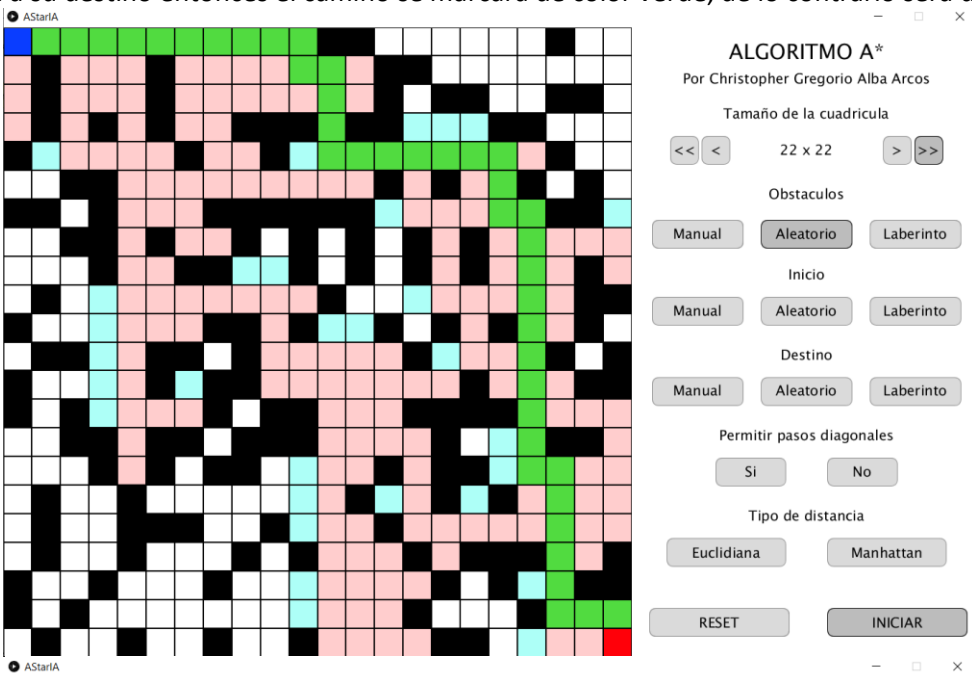
FUNCIONALIDAD DEL PROGRAMA:

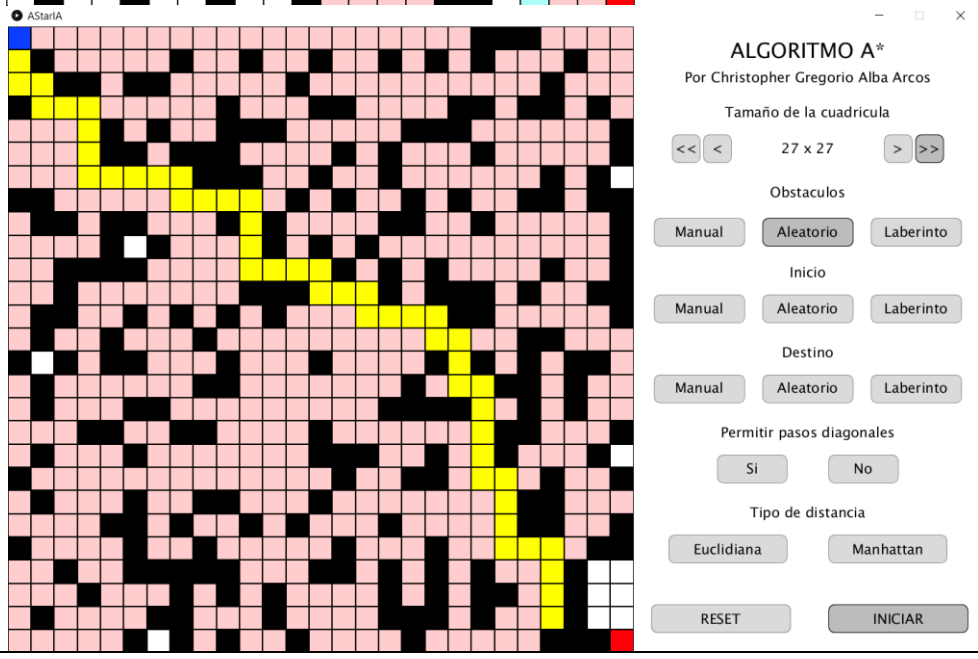
Se nos pidio la elaboracion de un programa que implementara el algoritmo de busqueda A* de forma visual, en el cual por medio de una cuadricula de 12x12 puediesemos agregar obstaculos y mover los puntos de inicio y destino. Sin embargo el diseño y funcion de este algoritmo me atrapo, desde el como influye cada uno de sus parametros hasta el querer ver hasta donde podia llegar tanto el procesamiento de mi computadora asi como yo en el ambito de programador, por lo que ademas de lo requerido he implementado funciones extras y modificacion de parametros en tiempo real, por lo que al inicio de este reporte se daran las caracteristicas estricatamente requeridas, hasta avanzar a aquellas que son meramente fuegos artificiales que me causaron mucha emocion al agregar.





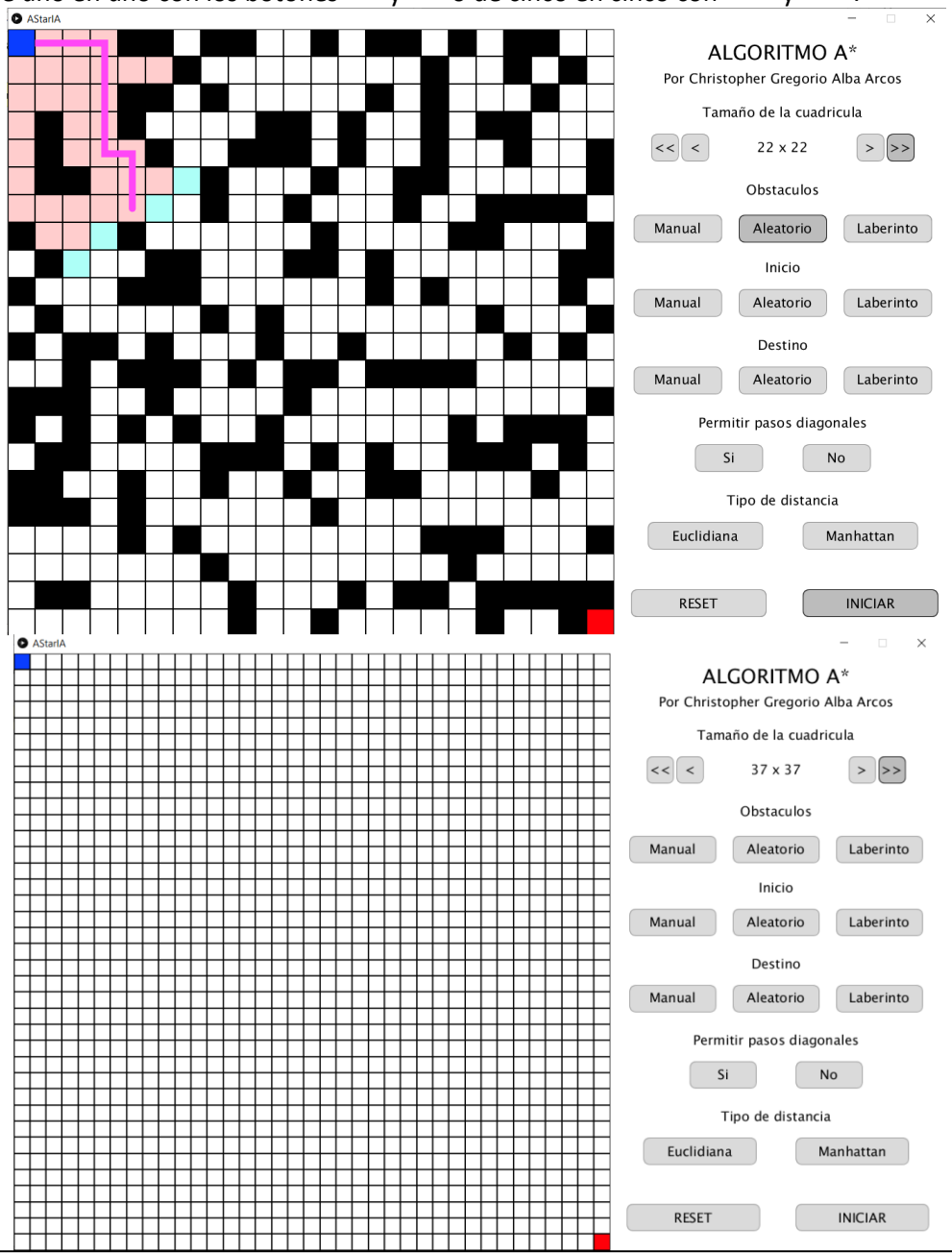
En las primeras imágenes notamos como la cuadricula inicial al momento de lanzar el programa es de 12x12, si oprimimos el boton de “manual” en la seccion de obstaculos y hacemos click sobre una celda esta se convertira en un obstaculos, de igual forma ocurre si oprimimos la tecla “o” creando el obstaculo en la celda sobre donde este el cursor, si volvemos a orpimir “o” sobre una celda que sea un obstaculo este pasara a estar libre.
Los obstaculos son marcados como celdas de color negro. Al empezar la ejecucion del algoritmo con el boton “iniciar” el algoritmo empezara su funcionamiento con los parametros y configuracion de la cuadricula que le hemos dados, si el algoritmo llega a su destino entonces el camino se marcara de color verde, de lo contrario sera de color amarillo.





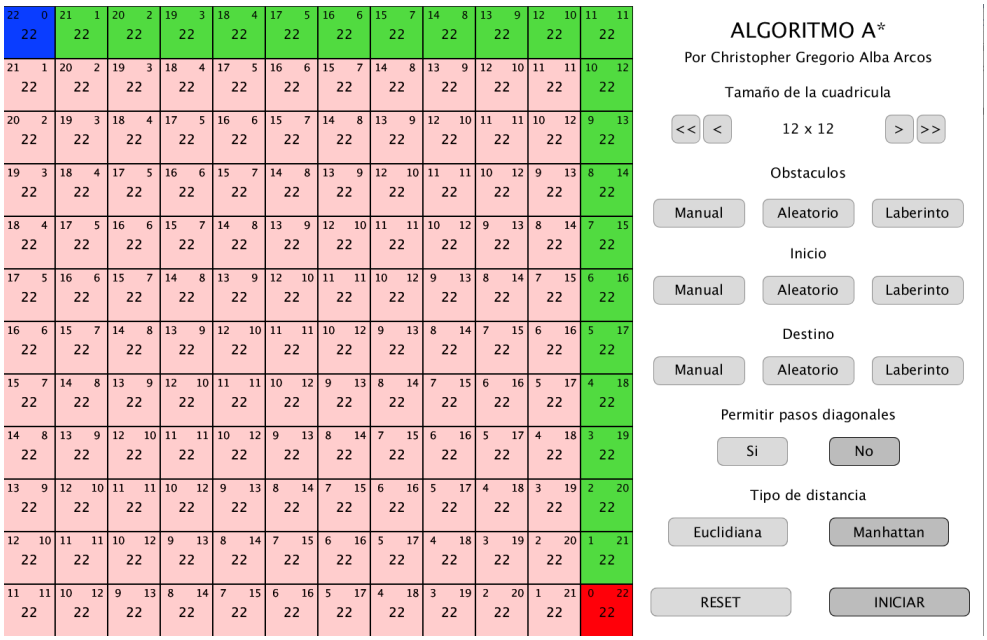
La funcion para modificar el punto de inicio o destino son identicas a la de los obstaculos, siendo las teclas “i” para el inicio y “d” para el destino.

Es aquí donde empezamos nuestro recorrido en cuanto a las funcionalidades extras que he agregado al programa, empezamos con la visualizacion del proceso del algoritmo, pues se nos mostrara el camino que se esta intentando actualmente en forma de una linea morada cuyo grosor depende del tamaño de cada celda, siendo el tamaño de las celdas a su vez dependiente de la cantidad de estas que se incluyen en la cuadricula. El tamaño de la cuadricula puede ser modificado de uno en uno con los botones “<” y “>” o de cinco en cinco con “<<” y “>>”.

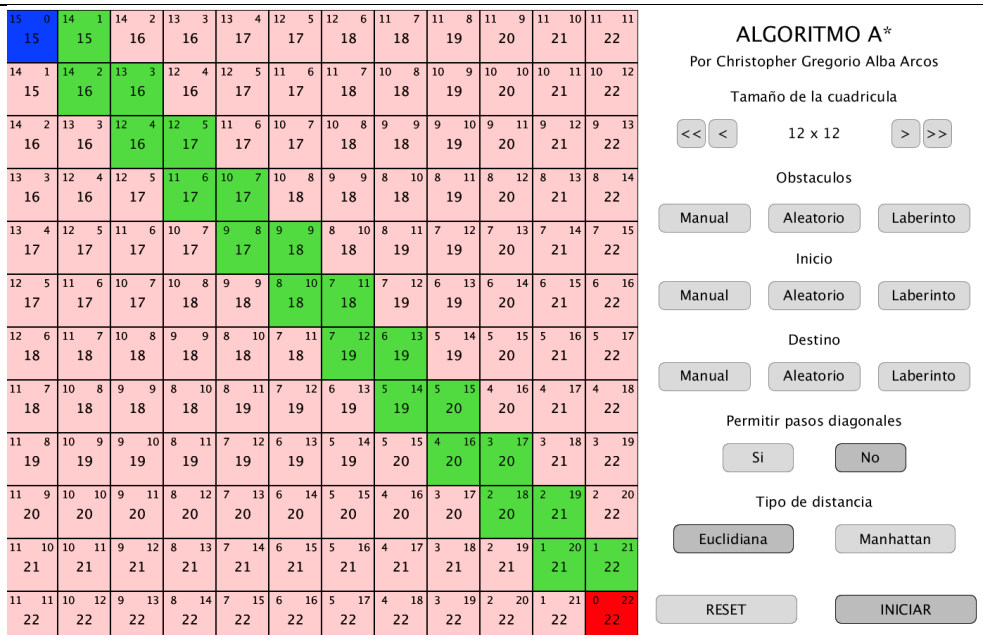


Cabe resaltar que tanto los obstáculos como los puntos de inicio y destino pueden ser generados en una posición aleatoria, en el caso de los obstáculos la forma en que se generan es relativamente sencilla pues el 30% de las celdas se volverán obstáculos de forma aleatoria. El inicio siempre aparecerá en la mitad izquierda y el destino en la mitad derecha, además de mantener el invariante de que ni el inicio ni el destino pueden ser obstáculos.
Se nos pidió que el programa solo permitiese pasos horizontales y verticales, y se nos dio la libertad de escoger el tipo de distancia que quisiéramos usar, sin embargo, tanto el solo tener una función de distancia como especialmente el no permitir el uso de pasos en vertical me parece un verdadero infortunio ya que nos cierra la puerta a comprender realmente como funciona el algoritmo A* y todas las posibles combinaciones que podemos obtener a partir de poder combinar estos parámetros; para pode apreciar de una mejor manera esto he agregado la funcionalidad de al oprimir la tecla “m” se muestren en el centro de cada celda visitada o calculada, la función f, mientras que en la esquina superior izquierda de esta se muestra la función h y en la esquina superior derecha la función g.

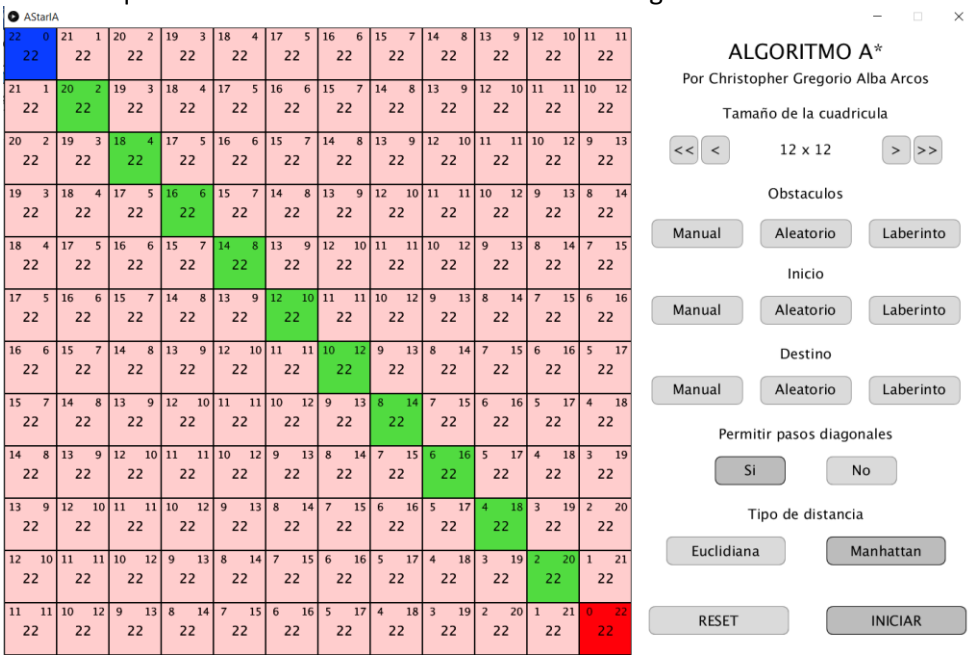
Podemos observar de forma clara como en el primer ejemplo al configurar una medida de distancia de tipo Manhattan o distancia de taxi, a la vez que no permitimos los pasos en diagonal tenemos que en cualquier punto de la cuadricula la función f vale lo mismo, y es que es lógico, al pensar que si se acerca al destino una unidad se aleje del inicio una unidad, por lo que la función f se mantiene en equilibrio, y con esto vemos que realmente al ser un campo libre de obstáculos absolutamente cualquier camino razonable (que no de vueltas) será el mejor camino.



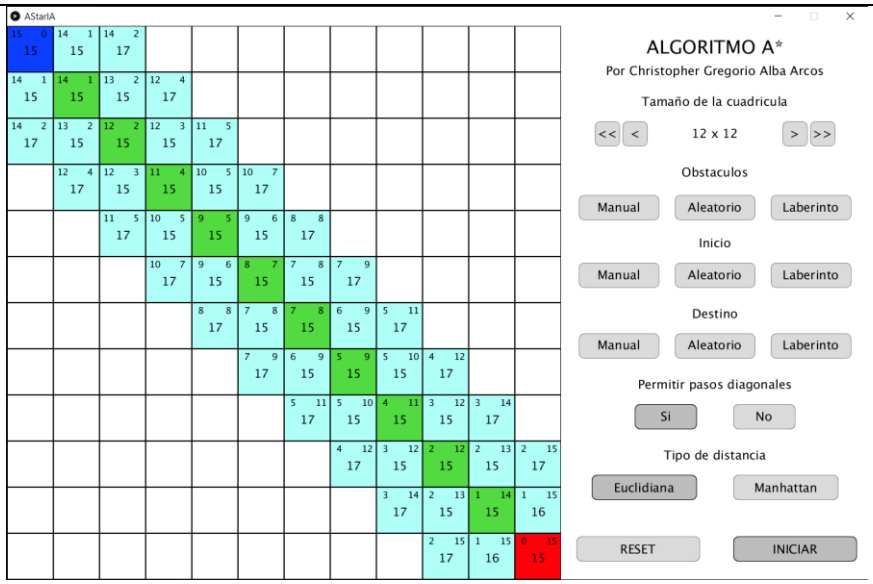
Si usamos la distancia euclidiana esta vez y seguimos sin permitir los pasos en diagonal obtendremos un resultado distinto y es que ahora nos encontramos con el hecho de que el teorema de Pitágoras se cumple y por lo tanto la hipotenusa será menor a la suma de los catetos o de forma mas simple, el camino mas corto entre dos puntos es la línea recta, a comparación de una que va con quiebres, por lo que iremos lo más diagonalmente posible, sin embargo debido a que no tenemos la posibilidad de ir en diagonal obtendremos este tipo de escalones. Los valores en las funciones irán aumentando de forma dispar ya que podría verse como los vectores desde los puntos de inicio y destino hasta el punto actual. El algoritmo aun tiene que investigar todo el campo ya que cuando g aumenta h disminuye si se va en cierta dirección y viceversa, lo que hace algunas celdas viables para buscar la ruta.



Si permitimos los pasos en diagonal con la distancia de Manhattan tendremos una combiancion de los dos anteriores casos, aun debemos de inspeccionar todas las celdas debido a como se comporta la distancia de Manhattan, pero veremos que la ruta ahora es trazada atraves de la diagonal.

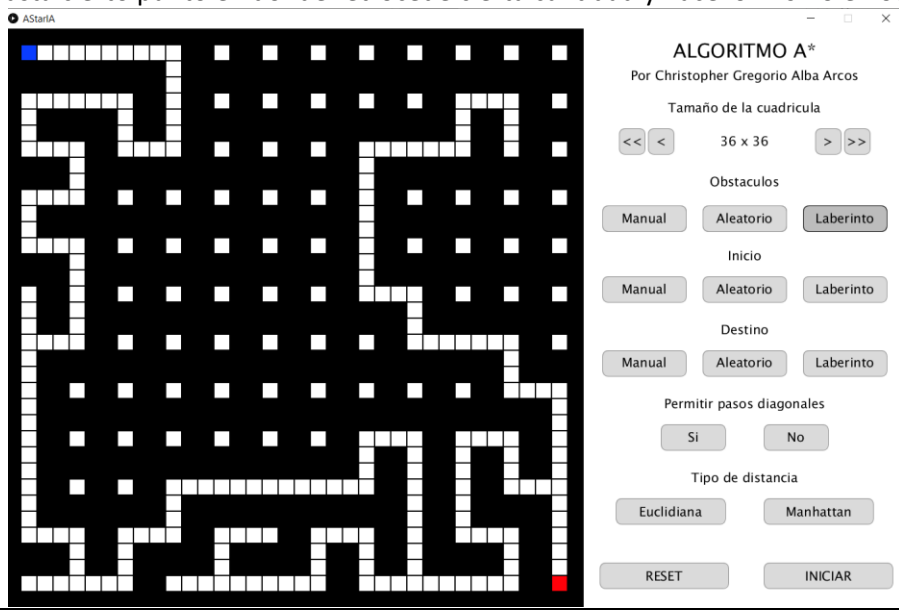


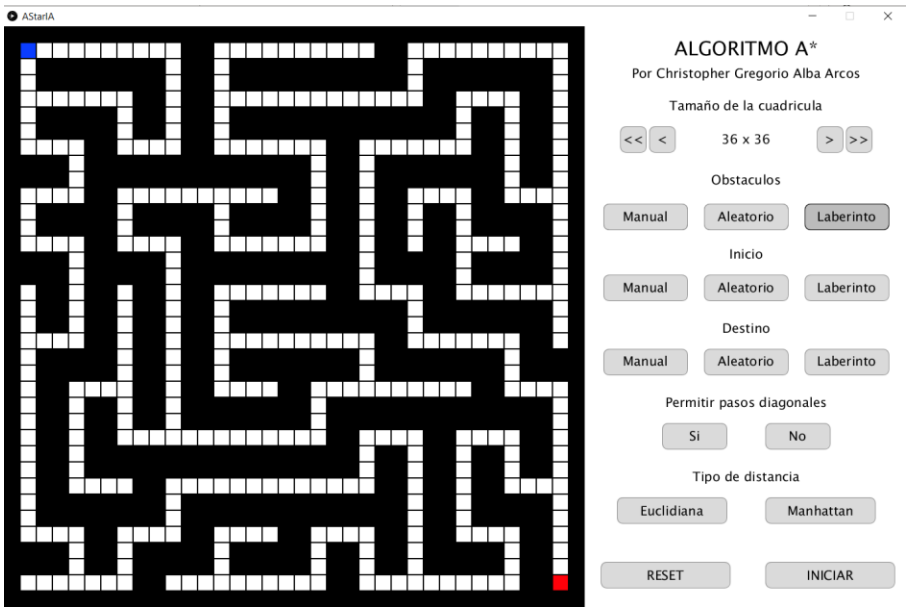
Ahora tendremos el mejor caso, en donde usamos la distancia euclidiana pero además permitimos los pasos en diagonal, por lo que podremos ir de la forma en que queremos según lo que nos dice nuestro “instinto” debemos ir. Por lo que encontramos la ruta de forma mas rápida, siendo que algunas celdas ni siquiera necesitan ser revisadas, pues nuestra función f no es menor a la que llevamos en la actual celda, por lo que no revisamos las demás celdas.



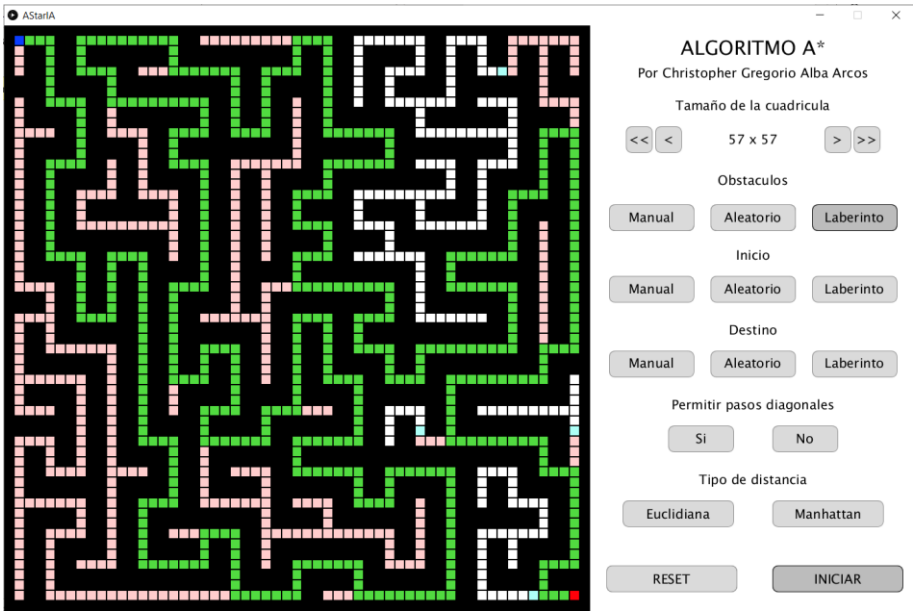
Y es justo aquí donde menciono otra de las implementaciones extras que realice al programa, en el algoritmo tendremos un conjunto abierto que será coloreado de azul bajo y un conjunto cerrado el cual será rojo bajo. El conjunto cerrado son las celdas que hemos evaluado y no necesitamos revisitar, y el abierto aquellos que quedan en "suspensión" por así decirlo, aun no siendo checados pero que en un futuro pueden serlo, en el momento de que la función f de celda en que estamos de nuestra ruta sea mayor que la de una del conjunto abierto.

El extra definitivo es la generación de un laberinto el cual posteriormente podremos resolver con el algoritmo. El laberinto es generado por medio de una implementación recursiva a través de la búsqueda en profundidad, para después hacer backtracking en donde cada tercer celda es una celda del laberinto que contiene una "pared" en cada dirección, las cuales son inicialmente obstáculos y el algoritmo va "excavando" en estos obstáculos uniendo los puntos de forma aleatoria hasta cierto punto en donde retrocede cierta cantidad y hace lo mismo en otra dirección.





Una vez que el algoritmo termina podremos ejecutar el algoritmo para resolver el laberinto que hemos creado, obviamente debido a que una celda de laberinto equivale a nueve celdas normales, el verdadero potencial del programa se alcanza al momento de que creamos una cuadricula relativamente grande, generamos un laberinto y lo resolvemos.



FACULTAD DE INGENIERÍA

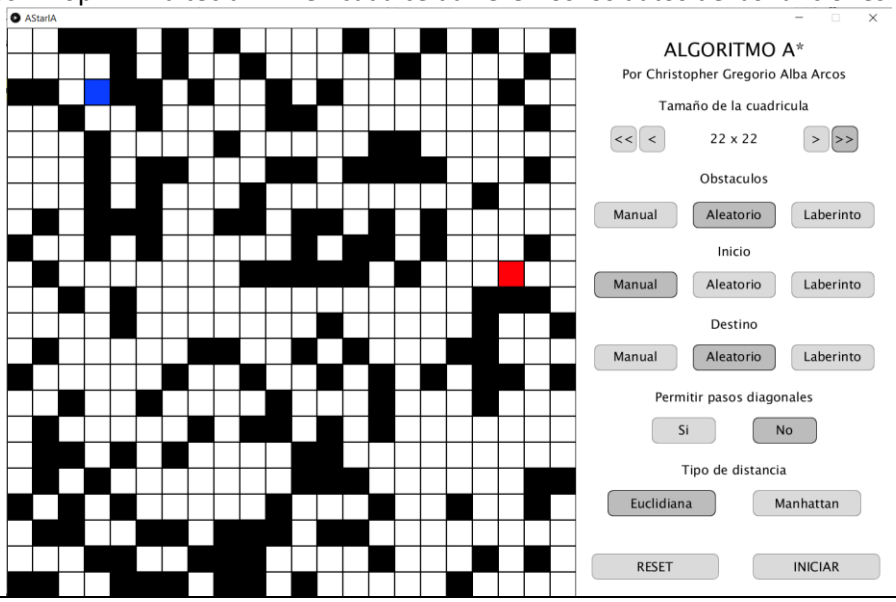
FORMATO
PRÁCTICAS DE LABORATORIO

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

INSTRUCCIONES DE USO:

El cómo funciona el programa ya ha sido descrito previamente, aunque es realmente simple y podemos resumirlo así:

- Cuadricula:
  - "<" y ">" aumentan el tamaño de la cuadricula de uno en uno
  - "<<" y ">>"aumentan el tamaño de la cuadricula de uno en uno
- Obstáculos:
  - Manual: Al hacer click en el botón "Manual" u oprimir la tecla "o" y hacer click sobre una celda de la cuadricula la volveremos un obstáculo, si hacemos click sobre una celda que ya es un obstáculo este se quitara. Esta opción puede ser utilizada sobre las otras.
  - Aleatorio: Se generarán obstáculos aleatorios por toda la cuadricula, un 30% de las celdas serán obstáculos y no se garantiza que exista una ruta entre el inicio y el destino. Esta opción puede ser utilizada sobre las otras, y sobre si misma lo que podría causar demasiados obstáculos.
  - Laberinto: Se genera un laberinto sobre la cuadricula el cual esta garantizado de tener solución amenos de que se le aplique sobre este alguna de las otras opciones. Esta opción sobrescribe las demás configuraciones por lo que debe de utilizarse de primera mano.
- Inicio/Destino:
  - Manual: Al hacer click en el botón "Manual" u oprimir la tecla "i"/"d" y hacer click sobre una celda de la cuadricula la volveremos el inicio/destino conservando la invariante de que estos no son obstáculos.
  - Aleatorio: El inicio/destino será colocado en un punto de menor/mayor a la mitad de la cuadricula.
  - Laberinto: El inicio/destino será colocado en la esquina superior izquierda/ inferior derecha donde se encuentra la primera celda del laberinto en caso de generar uno. O sea, el punto [1][1]/[n-2][n-2].
- Permitir pasos diagonales: Dando click en la opción si, serán permitidos los pasos en las 8 direcciones vecinas de cada celda, con la opción no solo se permitirán de forma horizontal y vertical.
- Tipo de distancia: Determina si se usara la distancia Euclidiana o la distancia de Manhattan al dar click en una.
- Iniciar: Al hacer click en este botón correremos el algoritmo con la configuración de parámetros y cuadricula.
- Reset: Con este botón reiniciaremos los parámetros y configuración inicial de la cuadricula.
- Mostrar datos: Al oprimir la tecla "m" en cada celda veremos los datos de las funciones h, g y f en ellas.



CODIGO RELEVANTE:

```
class Celda {
  int i;
  int j;
  float f = 0;
  float g = 0;
  float heuristica = 0;
  boolean visitado = false;
  boolean calculado = false;
  List<Celda> vecinos = new ArrayList<Celda>();
  Celda padre = null;
  boolean obstaculo = false;
  Celda(int i, int j) {
    this.i = i;
    this.j = j;
    obstaculo = false;
  }

  float getHeuristica() {
    return heuristica(this, destino, tipoDistancia);
  }

  float getF() {
    return getHeuristica()+g;
  }

  void mostrarCelda(color col) {
    fill(col);
    if (obstaculo) {
      fill(0);
    }
    stroke(0);
    strokeWeight(1);
    rect(i*w, j*h, w-1, h-1);

    if (calculado==true && mostrarDatos == true) {
      textAlign(CENTER, CENTER);
      fill(0);
      textSize(w/4);
      text(floor(getF()), i*w+w/2, j*h+h/2);
      textSize(w/5);
      text(floor(getHeuristica()), i*w+w/5, j*h+w/7);
```

```
    text(floor(g), i*w+w-w/5, j*h+w/7);
   }
  }

  Celda vecinoAleatorio() {
   ArrayList<Celda> vecinosLaberinto = new ArrayList<Celda>();
   Celda arriba, abajo, izquierda, derecha;
   try {
    arriba = cuadricula[i + 3][j];
   }
   catch(Exception e) {
    arriba = null;
   }
   try {
    abajo  = cuadricula[i - 3][j];
   }
   catch(Exception e) {
    abajo = null;
   }
   try {
    derecha = cuadricula[i][j + 3];
   }
   catch(Exception e) {
    derecha = null;
   }
   try {
    izquierda   = cuadricula[i][j - 3];
   }
   catch(Exception e) {
    izquierda = null;
   }
   if (arriba != null && !arriba.visitado) {
    vecinosLaberinto.add(arriba);
   }
   if (derecha != null && !derecha.visitado) {
    vecinosLaberinto.add(derecha);
   }
   if (abajo != null && !abajo.visitado) {
    vecinosLaberinto.add(abajo);
   }
   if (izquierda != null && !izquierda.visitado) {
    vecinosLaberinto.add(izquierda);
   }
```

```
  if (vecinosLaberinto.size() > 0) {
    int r = floor(random(0, vecinosLaberinto.size()));
    return vecinosLaberinto.get(r);
  } else {
    return null;
  }
 }

 void agregarVecinos(Celda[][] cuadricula, int tipo) {
  if (i < n - 1) {
    vecinos.add(cuadricula[i + 1][j]);
  }
  if (i > 0) {
    vecinos.add(cuadricula[i - 1][j]);
  }
  if (j < n - 1) {
    vecinos.add(cuadricula[i][j + 1]);
  }
  if (j > 0) {
    vecinos.add(cuadricula[i][j - 1]);
  }
  if (tipo == 1) {
    if (i > 0 && j > 0) {
      vecinos.add(cuadricula[i - 1][j - 1]);
    }
    if (i < n - 1 && j > 0) {
      vecinos.add(cuadricula[i + 1][j - 1]);
    }
    if (i > 0 && j < n - 1) {
      vecinos.add(cuadricula[i - 1][j + 1]);
    }
    if (i < n - 1 && j < n - 1) {
      vecinos.add(cuadricula[i + 1][j + 1]);
    }
  }
 }
}

float heuristica(Celda a, Celda b, int tipo) {
 float d = 0;
 if (tipo == 0) {
   d = abs(a.i - b.i) + abs(a.j - b.j);
 } else {
```

```
    d = dist(a.i, a.j, b.i, b.j);
  }
  return d;
}

void AStar() {
  if (openSet.size() > 0) {
    int minimo = 0;
    for (int i = 0; i < openSet.size(); i++) {
      if (openSet.get(i).f < openSet.get(minimo).f) {
        minimo = i;
      }
    }
    actual = openSet.get(minimo);
    actual.calculado = true;
    if (actual == destino) {
      iniciar = 0;
      noLoop();
    }
    openSet.remove(actual);
    closedSet.add(actual);
    List<Celda> vecinos = actual.vecinos;
    for (int i = 0; i < vecinos.size(); i++) {
      Celda vecino = vecinos.get(i);
      vecino.calculado = true;
      if (!closedSet.contains(vecino) && !vecino.obstaculo) {
        float tempG = actual.g + heuristica(vecino, actual, tipoDistancia);
        boolean nuevaRuta = false;
        if (openSet.contains(vecino)) {
          if (tempG < vecino.g) {
            vecino.g = tempG;
            nuevaRuta = true;
          }
        } else {
          vecino.g = tempG;
          nuevaRuta = true;
          openSet.add(vecino);
        }
        if (nuevaRuta) {
          vecino.f = vecino.g + vecino.getHeuristica();
          vecino.padre = actual;
        }
      }
```

```
    }
  } else {
    sinSolucion=true;
    iniciar = 0;
    noLoop();
    return;
  }
}
```

//Código para la generación del laberinto

```
void generarLaberinto() {
  actualLaberinto.visitado = true;
  Celda siguiente = actualLaberinto.vecinoAleatorio();
  if (siguiente != null) {
    siguiente.visitado = true;
    pila.add(actualLaberinto);
    quitarObstaculo(actualLaberinto, siguiente);
    actualLaberinto = siguiente;
  } else if (pila.size() > 0) {
    actualLaberinto = pila.remove(pila.size()-1);
  } else {
    tipoObstaculos = 0;
  }
}

void quitarObstaculo(Celda a, Celda b) {
  int x = a.i-b.i;
  int y = a.j-b.j;
  if (x==0) {
    if (y>0) {
      cuadricula[a.i][a.j-1].obstaculo=false;
      cuadricula[b.i][b.j+1].obstaculo=false;
    } else {
      cuadricula[a.i][a.j+1].obstaculo=false;
      cuadricula[b.i][b.j-1].obstaculo=false;
    }
  } else {
    if (x>0) {
      cuadricula[a.i - 1][a.j].obstaculo=false;
      cuadricula[b.i + 1][b.j].obstaculo=false;
    } else {
      cuadricula[a.i + 1][a.j].obstaculo=false;
```

```
    cuadricula[b.i - 1][b.j].obstaculo=false;
   }
  }

  for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
    cuadricula[i][j].agregarVecinos(cuadricula, 0);
   }
  }
}
```

BIBLIOGRAFÍA:

- Apuntes y clases del profesor
- Stout, B. (1996). Smart moves: Intelligent pathfinding. Game developer magazine, 10, 28-35.
- Niemczyk, R., & Zawiślak, S. (2020). Review of Maze Solving Algorithms for 2D Maze and Their Visualisation. In Engineer of the XXI Century (pp. 239-252). Springer, Cham.
- https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*
- https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*

Para la generación del laberinto:

- https://es.qaz.wiki/wiki/Maze_generation_algorithm#:~:text=Entonces%2C%20se%20puede%20considerar%20que,contribuyen%20al%20espacio%20de%20b%C3%BAsqueda.

Las bases de este código e inspiración fueron extraídas de los siguiente GitHub:

- https://github.com/CodingTrain/website/tree/main/CodingChallenges/CC_051_astar/P5

Para la generación del laberinto:

- https://github.com/CodingTrain/website/tree/main/CodingChallenges/CC_010_Maze_DFS/P5

siendo estos trabajos en JavaScript realizado por “The Coding Train” (https://thecodingtrain.com) con las librerías de P5.js