Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Inteligencia Artificial:

ALGORITMOS DE BÚSQUEDA. BÚSQUEDA INFORMADA

Samuel Martín Morales
(alu0101359526@ull.edu.es)

Cheuk Kelly Ng Pante
(alu0101364544@ull.edu.es)

Índice:

1. Introducción.

El propósito de esta práctica es la utilización de estrategias de búsqueda informadas para la planificación de trayectorias de un **coche autónomo** desde dos puntos **A** y **B**. El entorno se utilizará una matriz de dimensiones **MxN** constituido únicamente por celdas libres dónde el coche autónomo puede efectuar 4 posibles acciones de movimiento, una cada vez, desde la casilla **actual** a una de las **4-vecinas** (Norte, Sur, Este u Oeste).

1.1. Formulación del problema a resolver.

Tenemos un tablero de dimensiones **MxN** en principio todas las posiciones tienen el estado a *default*, al introducir el coche, se pone 1; al introducir la meta, se pone 3 y el estado 2 sería el camino generado por el algoritmo de búsqueda.

1.2. Algoritmo A*

El **algoritmo A*** es un algoritmo de **búsqueda informada**: a partir de un nodo de inicio específico de un gráfico, tiene como **objetivo** encontrar un camino hacia el nodo de **destino** dado que tiene el menor costo. Este algoritmo depende de dos funciones diferentes para formar su función de evaluación, que son los siguientes:

- g(n): Costo de la ruta desde el nodo de partida hasta el nodo n.
- h(n): Heurística de la ruta más barata desde n hasta el nodo objetivo.

Su forma de evaluación es la siguiente:

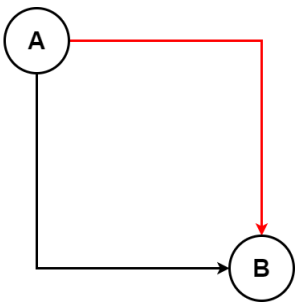
$$f(n) = g(n) + h(n)$$

1.3. Función heurística Manhattan

La función heurística **Manhattan** o distancia Manhattan es una métrica de distancia entre dos puntos en un espacio vectorial de N dimensiones. Es la suma de las longitudes de las proyecciones del segmento de línea entre dos puntos en los ejes de coordenadas.

La forma de evaluación es la siguiente:

$$h(n) = (x1 - x2) + (y1 + y2)$$

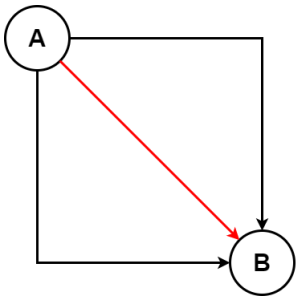


1.4. Función heurística Euclídea

La función heurística **Euclídea** o distancia Euclídea es la distancia más corta entre dos puntos en un espacio de N dimensiones. Se usa como una métrica común para medir la similitud entre dos puntos de datos.

Su forma de evaluación es la siguiente:

$$h(n) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$



2. Entregas.

Como se ha descrito anteriormente esta segunda práctica de la asignatura de 'Inteligencia Artificial' se basa en el empleo de estrategias de búsqueda informadas para la búsqueda del camino mínimo que conecte un punto **A** y un punto **B** que han sido especificados. Es por ello que, para facilitar el desarrollo e implementación del proyecto, se han solicitado varias entregas previas al proyecto final. Dichas entregas se pueden observar a continuación:

2.1. Entrega-1.

Para la primera entrega, se solicita un prototipo de proyecto que sea capaz de realizar la impresión de una matriz de MxN casillas, es decir, se solicita un simulador de entorno que permita imprimir un tablero.

Para dicho tablero, son especificadas una serie de dimensiones, las cuales corresponderá a la situación del problema. Una vez son especificadas las dimensiones del tablero, se marca el inicio del camino y su final.

Cabe destacar que para dicha primera entrega de prototipo, el camino mostrado por pantalla no corresponde el camino mínimo que debe de encontrar al presentar el proyecto final, sino, que debe de encontrar el camino que mejor se adapte para alcanzar el **punto A** y el **punto B**.

Es por ello que, para la implementación de este primer prototipo, se hace uso del **Lenguaje de Programación C++.** Permitiendo el desarrollo de dos clases que contendrán el grueso de la implementación del prototipo.

- La primera de estas clases es la clase '**Square**':

Esta, representa las distintas casillas que conforman el tablero. Permitiendo adquirir cada una de ellas un color distinto para la correcta impresión del tablero a través de la **terminal**. Además, en dicha clase se establece cuál es la casilla de **inicio** y cuál es la casilla de **destino**, de esta manera se puede implementar la búsqueda del camino de menor coste de forma más sencilla.

- La segunda de las clases es la clase '**Map**':

Dicha clase, representa el propio tablero del prototipo, es decir, contiene las distintas casillas del problema. De esta manera, se puede implementar la estratégia de búsqueda en el propio tablero del problema, pudiendo recorrer todas aquellas casillas que sean necesarias para la búsqueda del camino mínimo desde el punto **A** hasta el punto **B** y poder resolver el problema planteado.

Como se puede observar a continuación, se tiene la implementación de la clase '**Square**':

```
class Square {
  public:
    Square(size_t, size_t);
    ~Square();
    void setI(size_t);
    void setJ(size_t);
    void setState(size_t);
    void setfScore(double);
    void setgScore(double);
    void setCamefrom(size_t);
    size_t getI() const;
    size_t getJ() const;
    size_t getState() const;
    double getgScore();  /// Initial node cost to this node.
    double getfScore(); /// Estimated cost from initial node to target node.
    size_t getCamefrom();
    std::ostream& WriteSquare(std::ostream&);
  private:
    size_t i_;
    size_t j_;
    size_t state_;
    size_t camefrom_;
    double gScore_;
    double fScore_;
};
```

Por otra parte, la clase '**Map**' es:

```
class Map {
  public:
    Map() = default;
      Map(size_t rows, size_t cols, size_t start_row, size_t start_col, size_t
goal_row, size_t goal_col);
    ~Map();
    void setM(size_t rows);
    void setN(size_t cols);
    void setStart(size_t start_row, size_t start_col);
    void setGoal(size_t goal_row, size_t goal_col);
    void setHeuristicFlag(int heuristic_option);
    size_t getM() const;
    size_t getN() const;
    void setInitialState(size_t i, size_t j);
    void setGoalState(size_t i, size_t j);
    std::ostream& WhiteLine(std::ostream& os);
    void RouteSearch();
    void WriteMap(std::ostream &os);
    void StartPreparation();
    double Heuristic(size_t i, size_t j);
    std::vector<size_t> getNeighbors(size_t nodeId);
    void AStarAlgorithm();
  private:
    size_t M_;  // Number of rows
    size_t N_;  // Number of columns
    size_t initial_; // Initial state
    size_t goal_; // Goal state
    size_t start_row_;  // Start row
    size_t start_col_; // Start column
    size_t goal_row_;  // Goal row
    size_t goal_col_;  // Goal column
    Square **map_;  // Map
    bool flag_;
    int heuristic_option_;
    std::vector<size_t> _closedSet;
    std::vector<size_t> _openSet;
    std::vector<size_t> _path;
};
```

Finalmente, para esta primera entrega, se implementa el método '**RouteSearch()**' dentro de la clase '**Map**' que permite mostrar el camino para ir desde el punto **inicial** hasta el **final**, siendo esta función útil para comprobar el funcionamiento básico de este primer prototipo de proyecto.

Dicha función se puede observar a continuación:

```
void Map::RouteSearch() {
  if (start_row_ <= goal_row_) {
    if (start_row_ ==
        goal_row_) {
      if (start_col_ < goal_col_) {  /// Advance to the same row and to the right
        for (size_t i = start_col_ + 1; i <= goal_col_ - 1; i++) {
          map_[start_row_ * getN() + i]->setState(3);
        }
      } else {  /// Retreat to the same row and to the left
        for (size_t i = start_col_ + 1; i >= goal_col_ - 1; i--) {
          map_[start_row_ * getN() + i]->setState(3);
        }
      }
    } else {
      for (size_t i = start_row_ + 1; i <= goal_row_; i++) {
        map_[i * getN() + start_col_]->setState(3);
      }
      for (size_t i = start_col_ + 1; i <= goal_col_ - 1; i++) {
        map_[goal_row_ * getN() + i]->setState(3);
      }
    }
  } else {  /// Retreat case
    for (size_t i = start_row_ - 1; i >= goal_row_; i--) {
      map_[i * getN() + start_col_]->setState(3);
    }
    if (start_col_ < goal_col_) {  /// Ad
      for (size_t i = start_col_ + 1; i <= goal_col_ - 1; i++) {
        map_[goal_row_ * getN() + i]->setState(3);
      }
    } else {  /// Retreat to the same row and to the left
      for (size_t i = start_col_ + 1; i >= goal_col_ - 1; i--) {
        map_[goal_row_ * getN() + i]->setState(3);
      }
    }
```

```
  }
};
```

2.2. Entrega-2.

Para la segunda entrega de prototipo, se solicita la implementación de la estratégia de búsqueda seleccionada (**algoritmo Primero el Mejor ó A *)** por el programador, haciendo uso de dos tipos de evaluaciones heurísticas para la búsqueda del camino mínimo.

- La primera de ellas es la evaluación heurística haciendo uso de la distancia de **Manhattan,** como se puede observar a continuación, la implementación de dicha heurística para el proyecto es de manera:

```
int aux1 = map_[i]->getI() - map_[j]->getI(),
    aux2 = map_[i]->getJ() - map_[j]->getJ();
    cost = std::abs(aux1) + std::abs(aux2);
```

- La segunda de las heurísticas solicitada es la de **Euclides**, para la implementación de esta, se hace de la manera:

```
    cost = sqrt(((map_[i]->getI() - map_[j]->getI()) * (map_[i]->getI() -
map_[j]->getI())) + ((map_[i]->getJ() - map_[j]->getJ())*(map_[i]->getJ() -
map_[j]->getJ())));
```

Tras esto, se procede a la implementación del algoritmo de búsqueda **A*** (el seleccionado por los autores del proyecto), permitiendo obtener el camino mínimo que conecta una casilla **inicial** con la casilla **final** que ha sido especificada por el usuario de manera previa a la generación del tablero del simulador de entorno.

Es por ello que, finalmente, se realiza la implementación del algoritmo como se puede observar a continuación:

```
void Map::AStarAlgorithm() {
  int length_path = 0;
  int expanded_nodes = 0;
  StartPreparation();
  flag_ = false;

  while (!_openSet.empty()) {
    double lowestF = DBL_MAX;
    size_t current;
```

```
    for (auto it = _openSet.begin(); it != _openSet.end(); it++) {
      if (map_[*it]->getfScore() < lowestF) {
        lowestF = map_[*it]->getfScore();
        current = *it;
      }
    }

    if (current == goal_) {
      flag_ = true;
      break;
    } else {
      auto it = std::find(_openSet.begin(), _openSet.end(), current);
      _openSet.erase(it);
      _closedSet.push_back(current);
      std::vector<size_t> neighbors = getNeighbors(current);
      for (size_t i = 0; i < neighbors.size(); i++) {
        auto it = std::find(_closedSet.begin(), _closedSet.end(), neighbors[i]);
        if (it == _closedSet.end()) {
          double tentativeGScore = map_[current]->getgScore() + Heuristic(current,
neighbors[i]);
          auto it = std::find(_openSet.begin(), _openSet.end(), neighbors[i]);
          if (it == _openSet.end()) {
            expanded_nodes++;
            _openSet.push_back(neighbors[i]);
          } else if (tentativeGScore >= map_[neighbors[i]]->getgScore()) {
            continue;
          }
          map_[neighbors[i]]->setCamefrom(current);
          map_[neighbors[i]]->setgScore(tentativeGScore);
          map_[neighbors[i]]->setfScore(tentativeGScore + Heuristic(neighbors[i],
goal_));
        }
      }
    }
  }
  if (flag_) {
    _path.push_back(goal_);
    size_t current = map_[goal_]->getCamefrom();
```

```
    while (current != initial_) {
      length_path++;
      _path.push_back(current);
      size_t previous = map_[current]->getCamefrom();
      map_[current]->setState(3);
      current = previous;
    }
    std::cout << Green_ << "\n\n¡Hay un camino al destino!" << std::endl;
    std::cout << Cyan_ << "\nLongitud del camino mínimo: " << length_path;
    std::cout << Cyan_ << "\nNodos expandidos: " << expanded_nodes << RESET <<
std::endl << std::endl;;
  } else {
    std::cout << Red_ << "¡No hay camino hacia el destino!" << RESET << std::endl;
  }
  return;
}
```

3. Resultados Finales.

Tras las dos entregas de prototipos, se obtiene finalmente al algoritmo de búsqueda informada **A***, que permite encontrar el camino mínimo entre un punto **A** y un punto **B** que ha sido especificado previamente.

3.1. Resultados Heurística **Manhattan**.

Para la comprobación de los distintos resultados, se realiza el testeo de distintos tamaños de tablero y la comprobación de la búsqueda aplicada a cada uno de los tamaños.

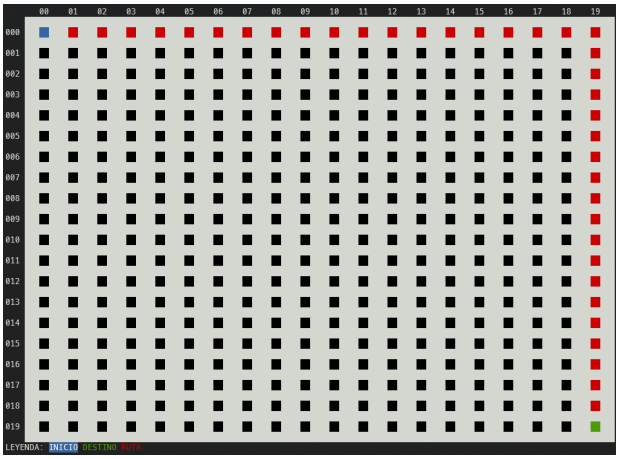


Figura 1: Tablero de tamaño 20x20.

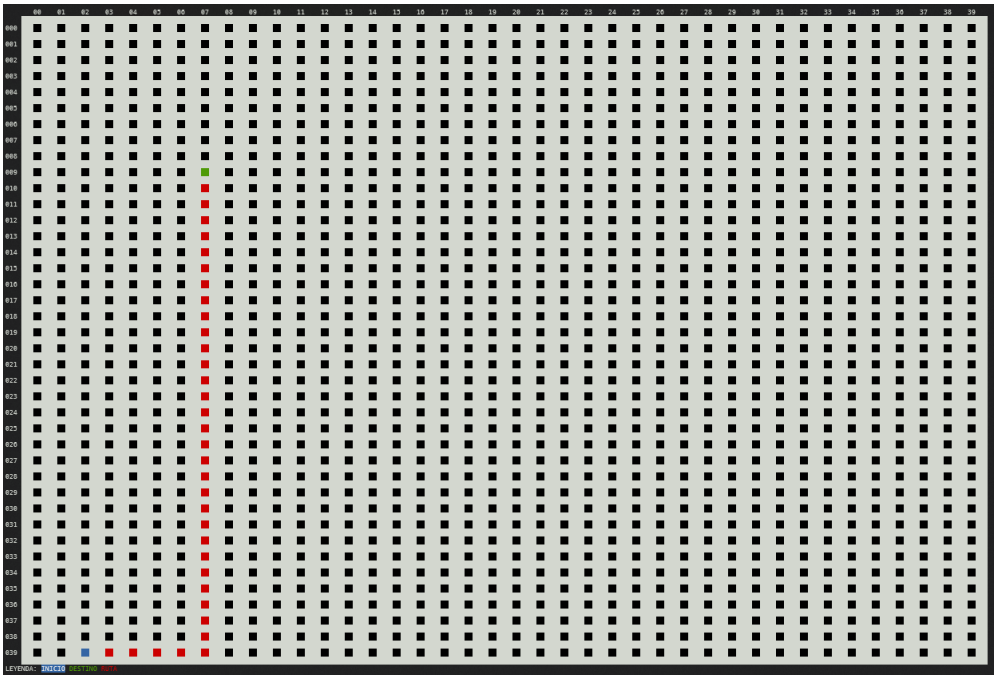


Figura 2: Tablero de tamaño 40x40.

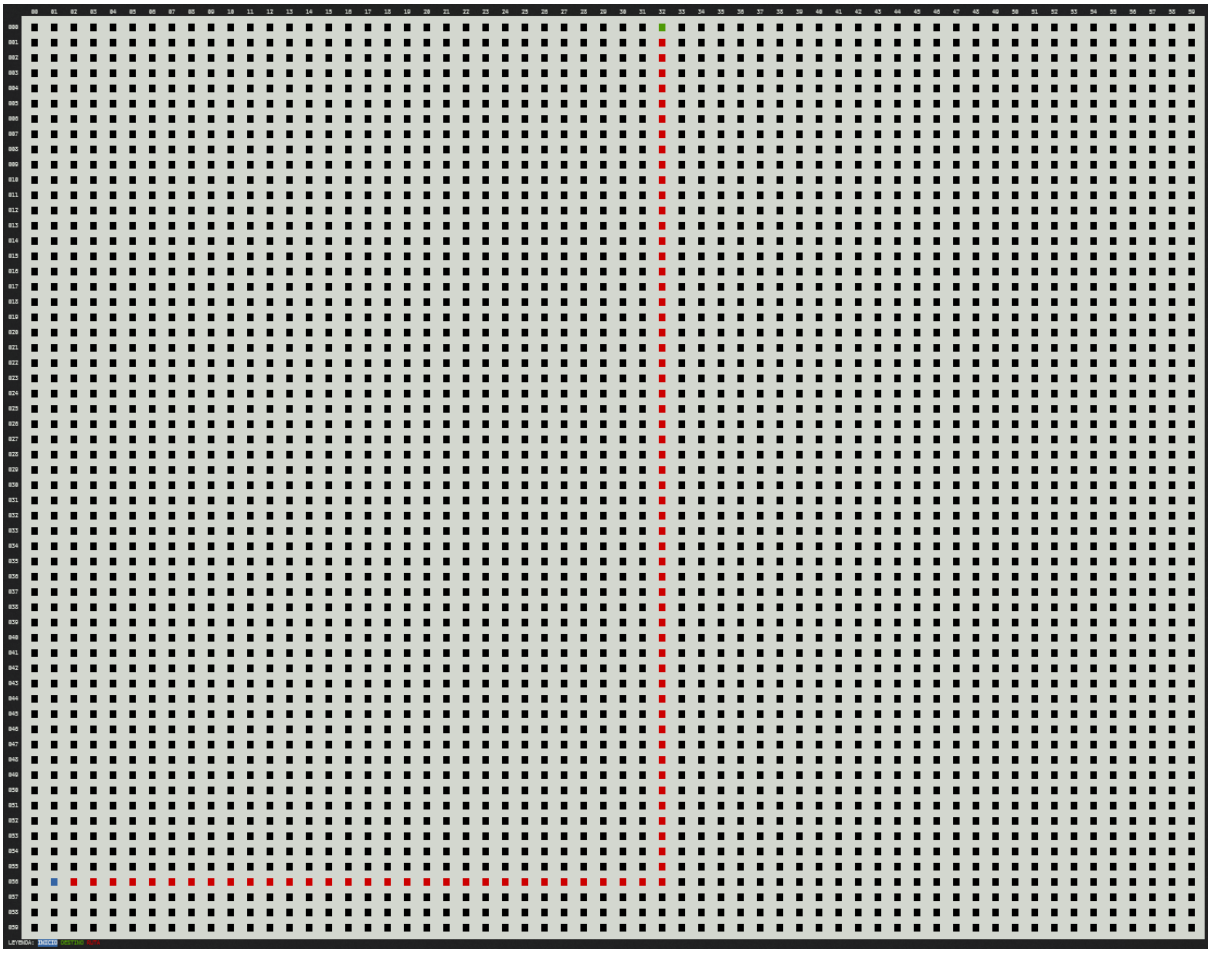


Figura 3: Tablero de tamaño 60x60.

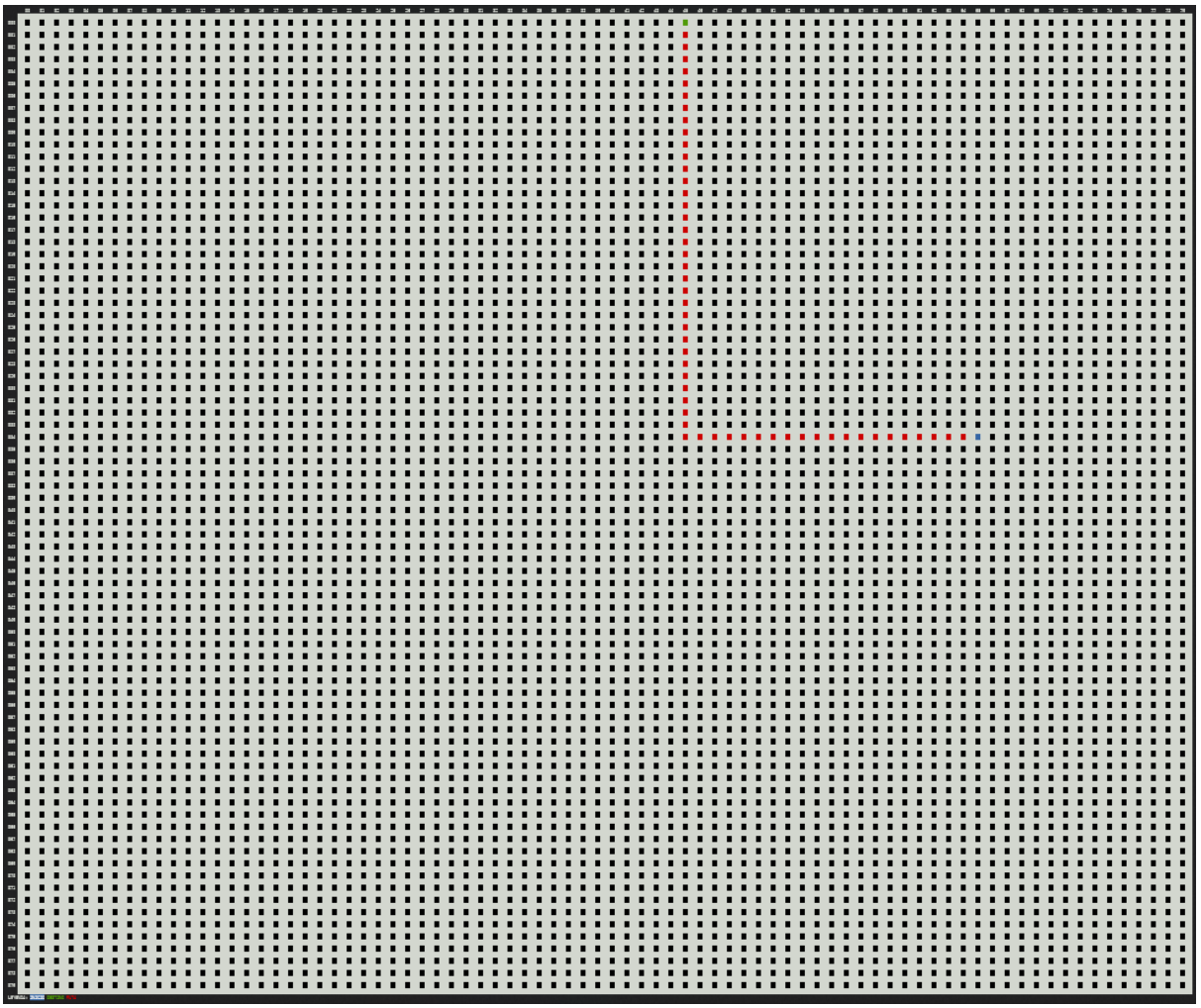


Figura 4: Tablero de tamaño 80x80.

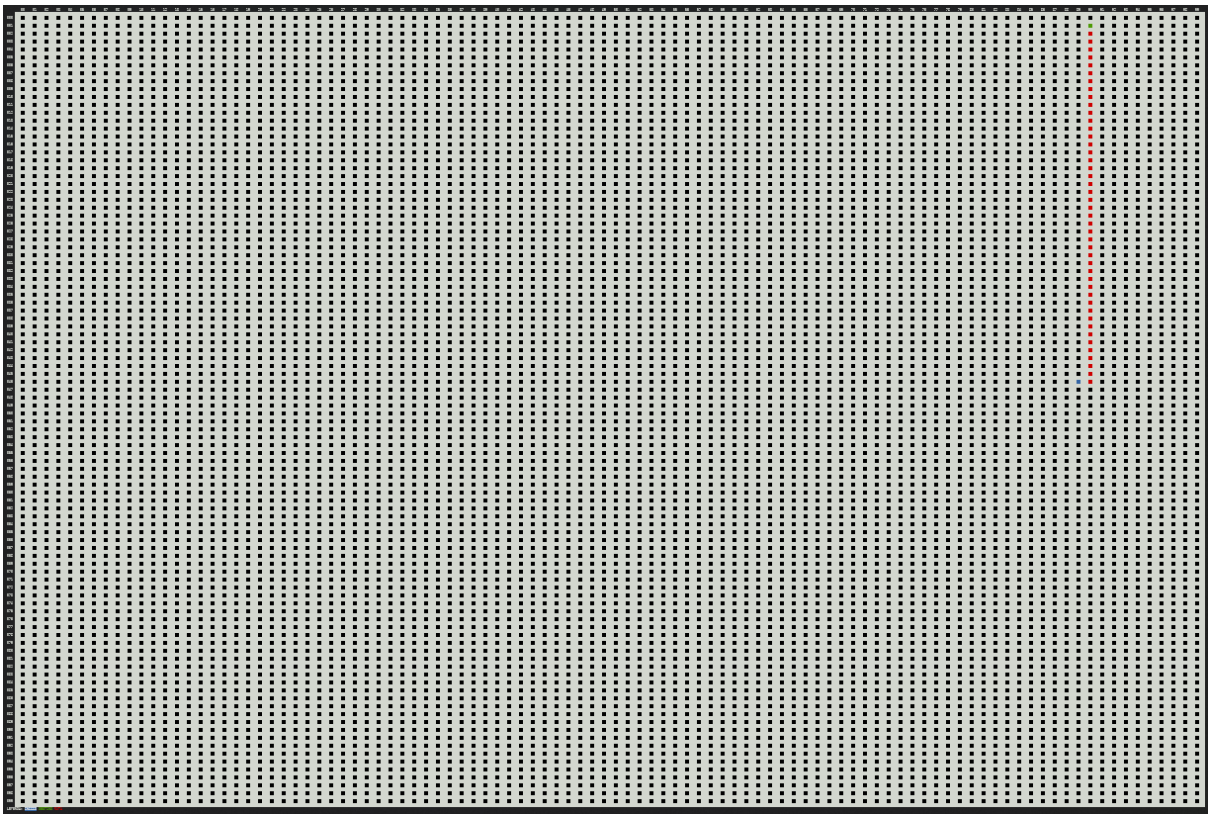


Figura 5: Tablero de tamaño 100x100.

3.2. Resultados Heurística **Euclides**.

Para la comprobación de los distintos resultados, se realiza el testeo de distintos tamaños de tablero y la comprobación de la búsqueda aplicada a cada uno de los tamaños.

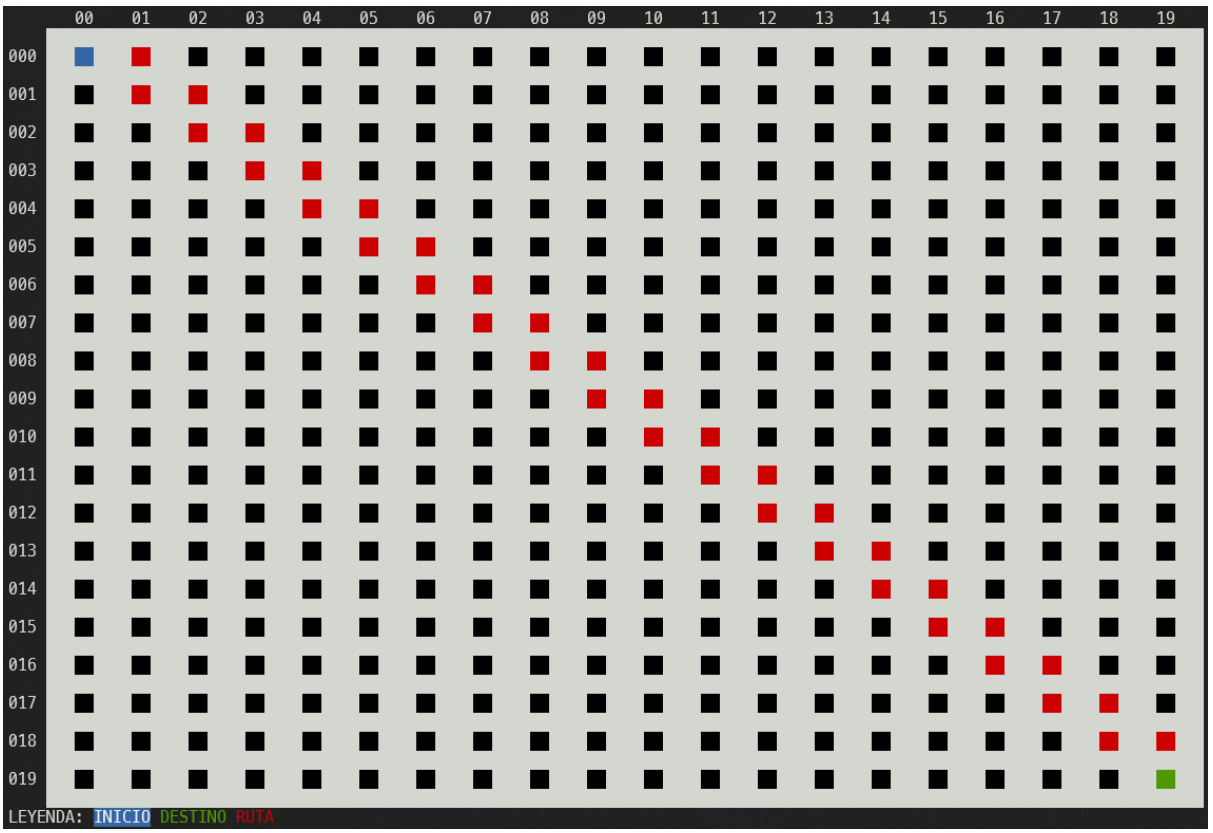


Figura 6: Tablero de tamaño 20x20.

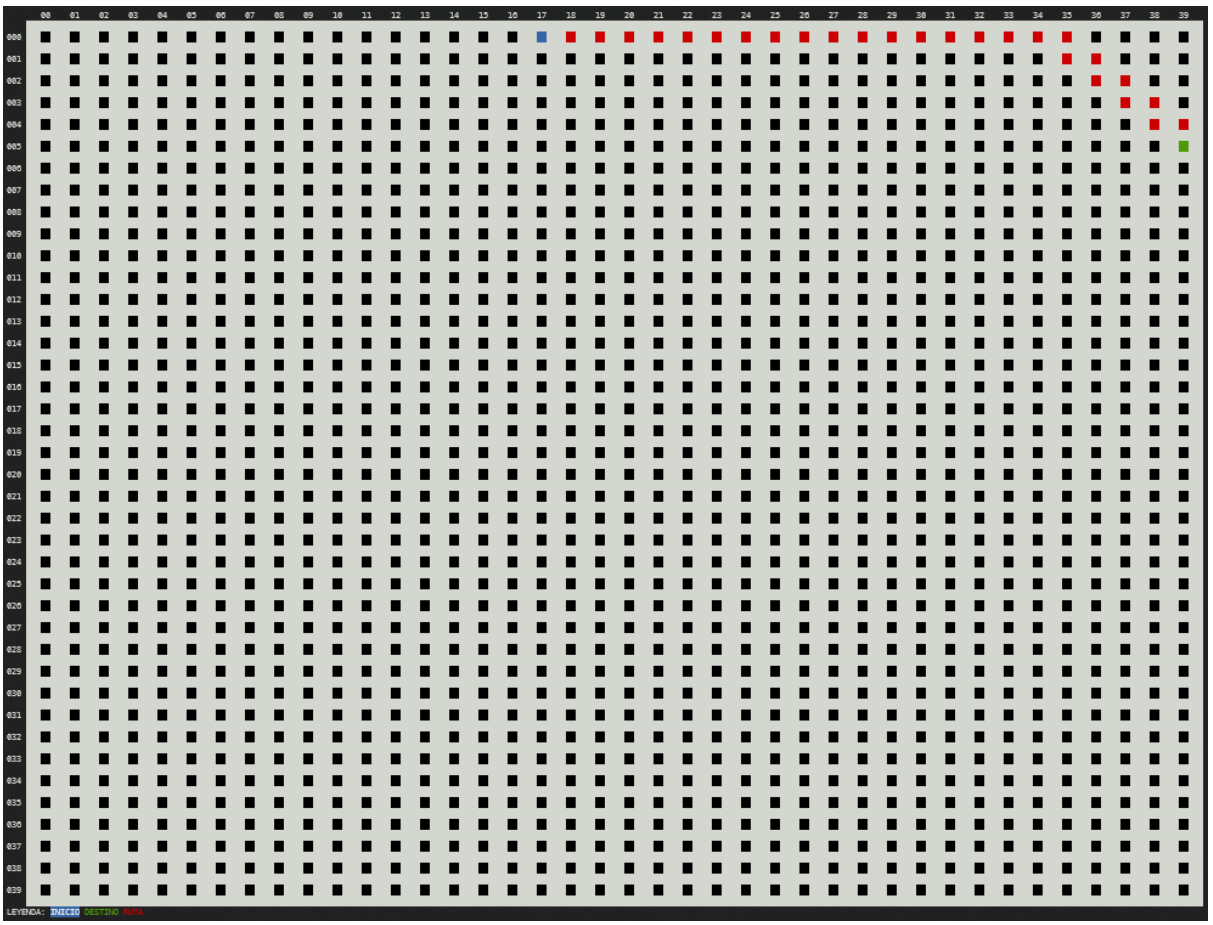


Figura 7: Tablero de tamaño 40x40.

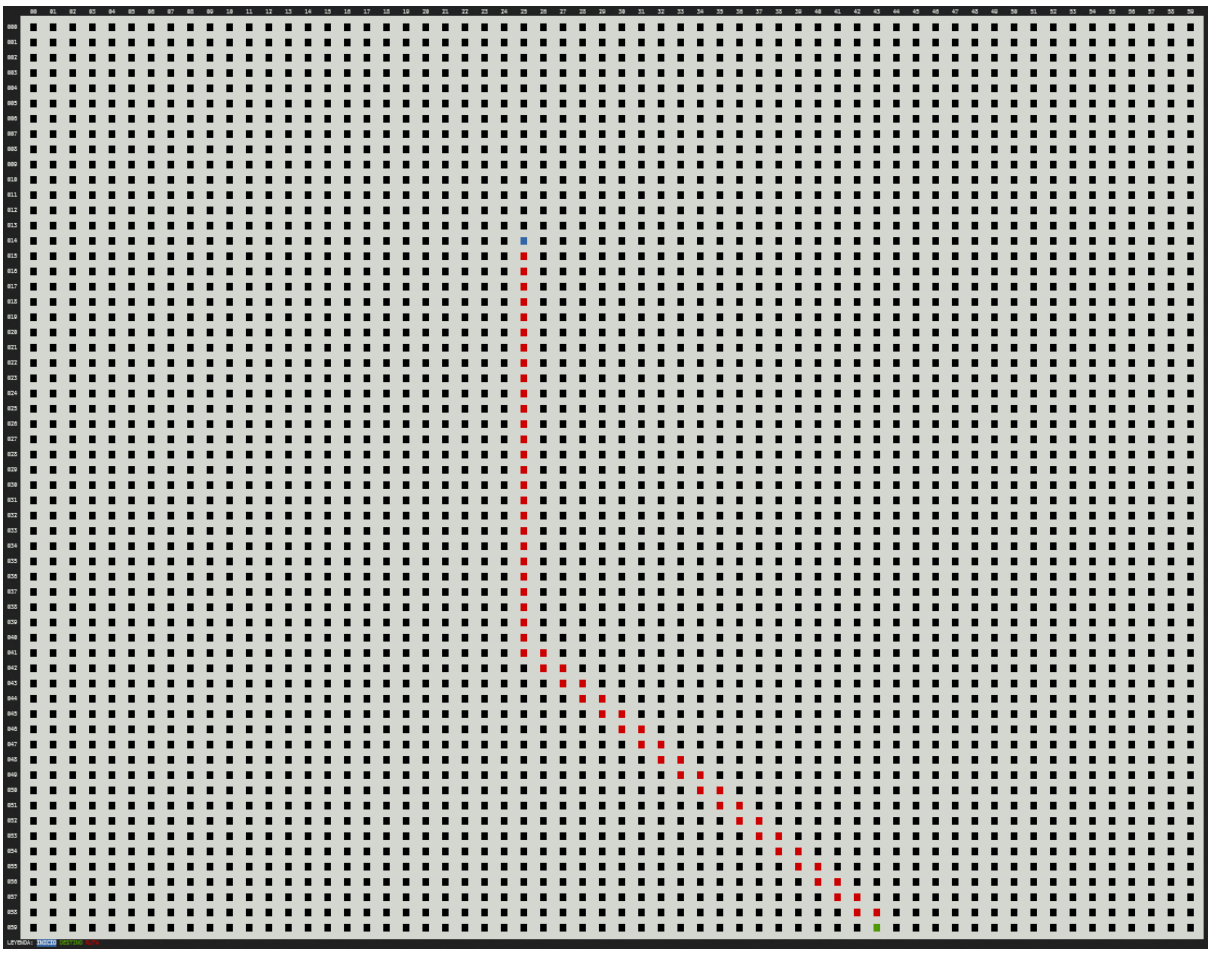


Figura 8: Tablero de tamaño 60x60.

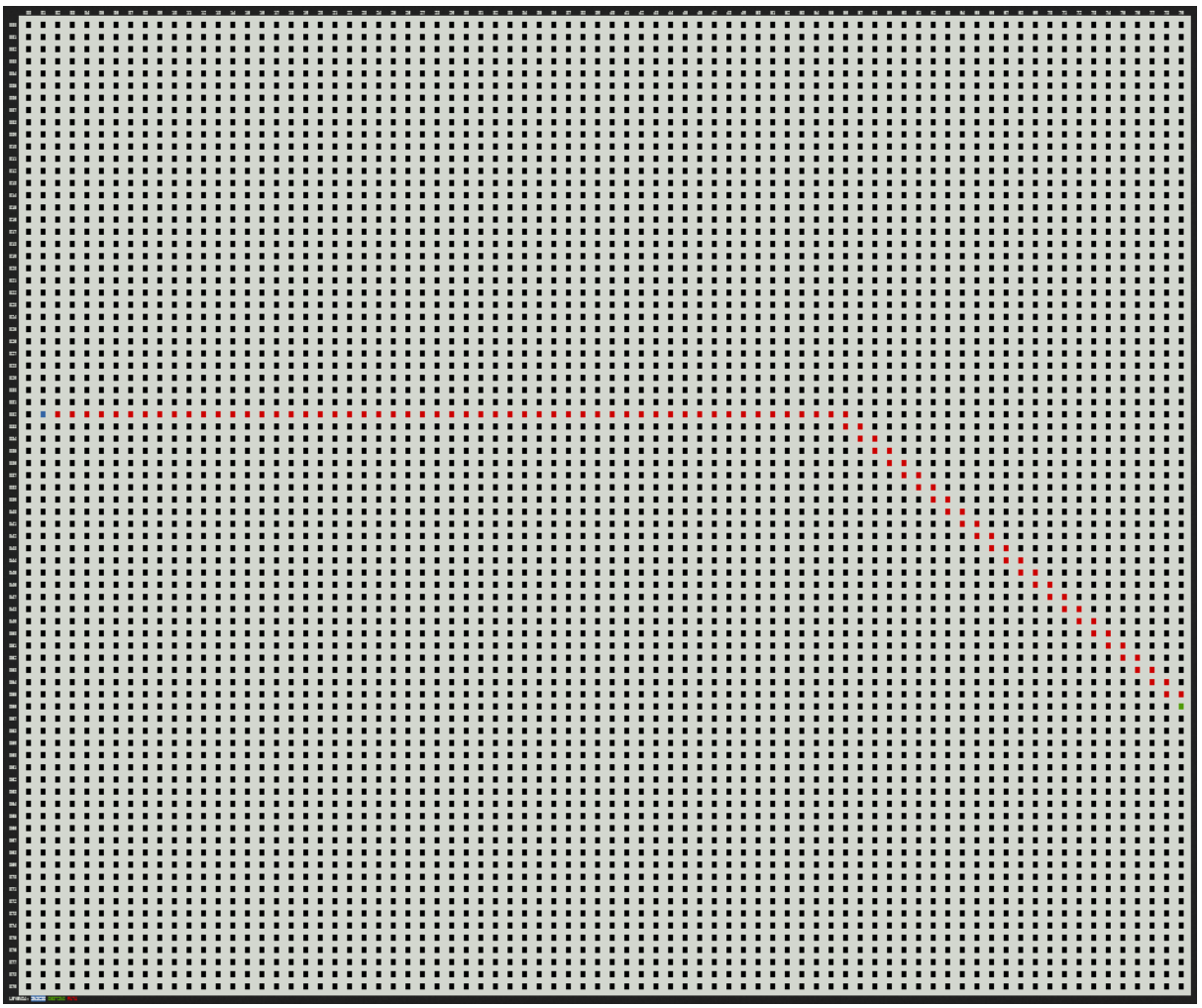


Figura 9: Tablero de tamaño 80x80.

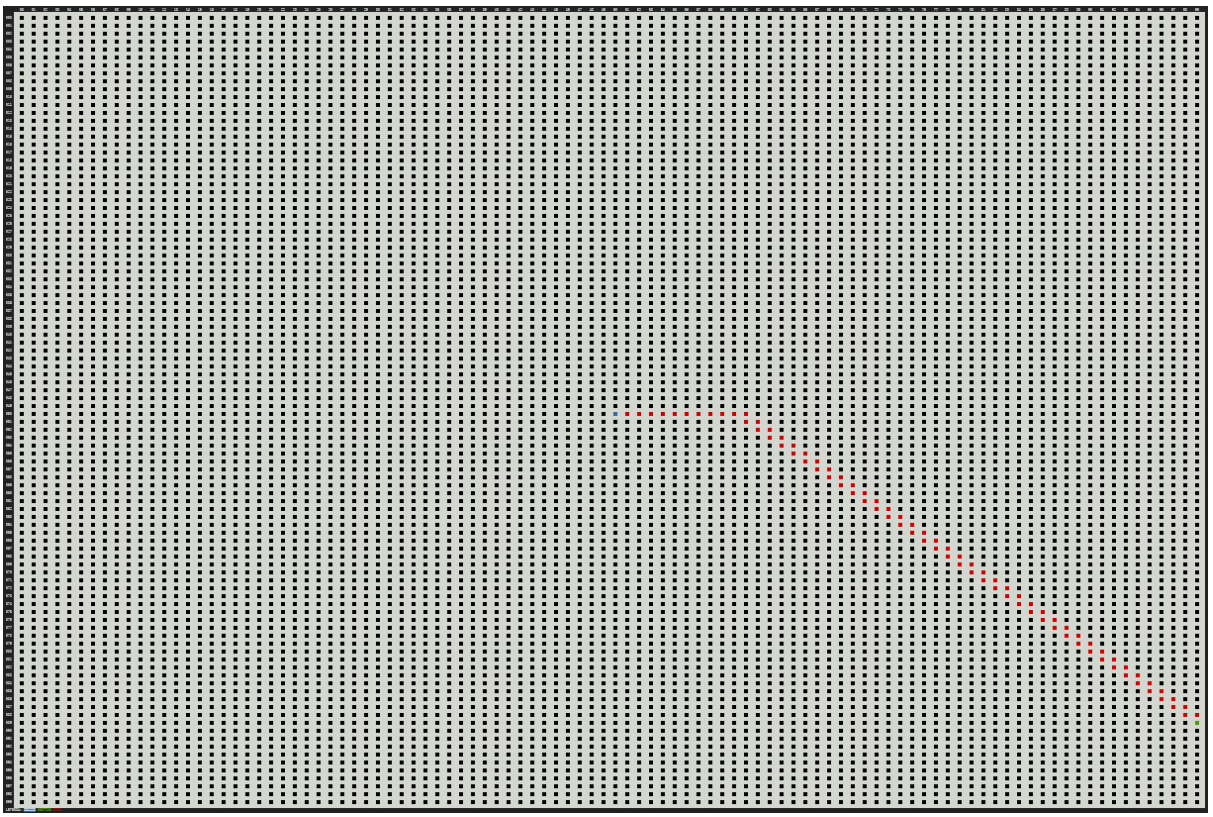


Figura 10: Tablero de tamaño 100x100.

4. Conclusión.

Para concluir, este proyecto, nos ha permitido conocer un poco más el funcionamiento de los algoritmos de búsqueda informada, además, de poder aprender sobre las distintas heurísticas que pueden ser aplicadas a dichos algoritmos de búsqueda, ya que, como se ha podido observar anteriormente, se ha aprendido a diferenciar entre la heurística **Manhattan** y **Euclídea**, permitiendo observar que en ciertos casos una se presenta como más óptima frente a la otra.

Por otra parte, este proyecto nos ha permitido además adquirir nuevos conocimientos en programación, para nuestro caso, en programación haciendo uso de **C++**, ya que, hemos podido investigar nuevas herramientas para la impresión de mejor manera a través de la terminal, o descubrir nuevas estructuras dentro del propio lenguaje, que nos han permitido implementar el proyecto final de mejor manera.

5. Referencias.

1. https://en.wikipedia.org/wiki/A*_search_algorithm
2. https://iq.opengenus.org/manhattan-distance/
3. https://iq.opengenus.org/euclidean-distance/