Instituto Tecnológico de Costa Rica

Inteligencia Artificial

Tarea 1 - Algoritmos de búsqueda

Authors:
Carlos Jenkins - 200769999
Pablo Rodriguez - 200767344

14 de octubre de 2012

Índice

Índice de cuadros

Índice de figuras

1. Descripción del problema

Los algoritmos de búsqueda se utilizan constantemente en aplicaciones de IA tales como robótica. En estos contextos, sin embargo, es importante adaptar dichos algoritmos a un entorno dinámico. Es común, por ejemplo, que el agente tenga que responder en un marco de tiempo limitado, que tenga que movilizarse físicamente para encontrar las rutas, y que en general se encuentre trabajando en linea. En la presente tarea se describe la implementación de tres algoritmos clásicos de búsqueda:

- Greedy (Best-First-Search).
- A* (A estrella).
- Jump points search.

A continuación se presenta la descripción de la solución (modificaciones al servidor gráfico, funciones generales y descripción de los algoritmos de búsqueda) y el análisis de resultados de la implementación.

2. Descripción de la solución

2.1. Modificaciones al servidor gráfico

Para el presente proyecto se utilizó una cuadrícula gráfica desarrollada por el profesor en el archivo `board.erl`. El API de dicha cuadrícula está presente en la especificación del proyecto y consiste en:

```
> board:start().
```

Que inicia la interfaz gráfica y permite configurar el entorno, en particular guardar y salvar archivos.

```
> board ! {get_pos, self()}.
```

Se obtiene la posición del agente en un tupla `{X, Y}`.

```
> board ! {move, {X, Y}}.
```

Mueve el agente a la posición `{X, Y}`. Dicha posición debe estar en el *Fringe* del algoritmo. Útil para los algoritmos Greedy y A* pero no para el Jump Points.

```
> board ! {get_neigbors, self()}.
```

Obtiene los vecinos del nodo actual en el que se ubica el agente. Devuelve una tupla de dos listas, la primera contiene las tuplas de los vecinos y la segunda la heurística euclidiana asociada a cada uno. Es de la forma:

```
{[{X1, Y1}, {X2, Y2}], [0.1, 0.2]}
```

Para poder implementar los algoritmos del presente proyecto se agregaron dos llamadas al servidor gráfico:

```
> board ! {jump, {X, Y}}.
```

Mueve al agente a la posición `{X, Y}`. La posición no necesariamente debe estar en el *Fringe*. Es utilizada por el algoritmo Jump Points.

```
> board ! {get_finish, self()}.
```

Obtiene una tupla `{X, Y}` con la posición del destino. Es utilizado por todos los algoritmos para saber cuando deben terminar.

Además, se tuvo que modificar la función de la cuadrícula `neighbors()` para que devolviera el punto de finalización en los vecinos, pues anteriormente lo excluía lo que dificultaba los algoritmos Greedy y A* para saber cuando han llegado al destino. La función modificada quedó de la siguiente forma:

```
neighbors(_,_,[]) -> [];
neighbors(Board,Cell={R,C},[{OR,OC}|Rest]) ->
    Pos = calcPos(R+OR,C+OC),
    if
    Pos == invalid -> neighbors(Board, Cell, Rest);
    true ->
        case element(Pos, Board) of
        ?EMPTY ->
            [{R+OR,C+OC}|neighbors(Board, Cell, Rest)];
        ?FINISH ->
            [{R+OR,C+OC}|neighbors(Board, Cell, Rest)];
        _ ->
            neighbors(Board, Cell, Rest)
        end
    end.
```

2.2. Funciones de utilidad general

Éstas funciones son funciones de utilidad general que se implementaron para el buen funcionamiento de los algoritmos.

Función `wait`:

Ésta función realiza una espera de `Mill` milisegundos antes de continuar en la ejecución. Parámetros:

- `Mill` milisegundos, es un valor entero.

Código:

```
wait(Mill) ->
    receive
    after Mill -> ok
    end.
```

Función `minimum`:

Ésta función se encarga de obtener una tupla con el valor mínimo y la posición del mismo una lista de números. Parámetros:

- `[Head|Tail]` es una lista no vacía de valores numéricos.

Código:

```
minimum([]) -> error;
minimum([Min]) -> {Min, 1};
minimum([Head|Tail]) -> minimum(Head, Tail, 1, 2).
minimum(Min, [Head|Tail], Index, CIndex) ->
    if
        Head < Min ->
            minimum(Head, Tail, CIndex, CIndex + 1);
        true ->
            minimum(Min, Tail, Index, CIndex + 1)
    end;
minimum(Min, [], Index, _) -> {Min, Index}.
```

Función `index_of`:

Esta función se encarga de obtener el índice en el que se encuentra un elemento en una lista dada, o en caso de no encontrarse, devuelve que no lo encontró. Parámetros:

- `Item` el elemento que va a buscar en la lista.

- `List` la lista en la que va a buscar el elemento.

Código:

```
index_of(Item, List) -> index_of(Item, List, 1).
index_of(_, [], _) -> not_found;
index_of(Item, [Item|_], Index) -> Index;
index_of(Item, [_|Tail], Index) -> index_of(Item, Tail, Index + 1).
```

Función `delete_nth`:

Esta función se encarga de eliminar el valor de una lista que se encuentra en la posición dada. En caso de que la posición solicitada sea mayor al largo de la lista, retorna que no encontró la posición. Parámetros:

- `Pos` la posición en la lista que se desea eliminar.
- `[H|T]` la lista en la cual de desea eliminar el elemento.

Código:

```
delete_nth(_,[]) -> not_found;
delete_nth(Pos,[H|T]) ->
    if
        Pos == 1 -> T;
        true -> lists:append([H], delete_nth(Pos - 1, T))
    end.
```

Función `euclidean`:

Esta función se encarga de obtener la distancia euclidiana que hay entre la diferencia de dos puntos. Parámetros:

- `Dx` es la diferencia de los valores de X de los dos puntos.
- `Dy` es la diferencia de los valores de Y de los dos puntos.

Código:

```
euclidean(Dx, Dy) ->
    math:sqrt((Dx * Dx) + (Dy * Dy)).
```

2.3. Descripción de algoritmos

A continuación se describe el funcionamiento básico y la estructura de los algoritmos implementados.

2.3.1. Greedy

El algoritmo Greedy consiste en realizar la búsqueda, en el tablero, bajo el objetivo de buscar la menor distancia desde donde estoy parado hasta el objetivo. Este algoritmo se implemento en dos funciones, una que obtiene los datos necesarios para el procedimiento y la otra que realiza el procedimiento.

Función `greedy()`**:**

Esta es la función con la que se preparan las estructuras necesarias y se llama al procedimiento. Para llamar al procedimiento es necesario el llevar el punto final al que se desea llegar y dos listas vacías, estas para poder realizar un backtracking. Parámetros:

- ninguno.

Función `greedy_proc({Fi, Fj}, {FringeCell, FringeHeu})`**:**

Esta es la función encargada de realizar la logística del procedimiento. Lo primero que hace es ver la posición en la que se encuentra, esto para verificar si ya se encuentra en la posición final. Si no se encuentra en la posición final solicita los vecinos, a los cuales se pueda mover, de la posición en la que se encuentra. Con estos vecinos, busca cual de ellos posee la menor heurística, valor por el cual se guía, para avanzar hacia esa posición. En caso de no obtener vecinos, revisa en las lista de `FringeCell` y `FringeHeu` para encontrar otra posibilidad de camino, en caso de que estas listas se encuentren vacías retorna que no es posible encontrar un camino valido. Parámetros:

- {`Fi, Fj`} punto objetivo.
- `FringeCell` lista de puntos en el tablero a los cuales es posible visitarse.
- `FringeHeu` lista de las heurísticas de los nodos en la lista `FringeCell` en su mismo orden.

2.3.2. A*

El algoritmo A* consiste en realizar la búsqueda, en el tablero, bajo el objetivo de buscar la menor distancia desde el inicio de la búsqueda hasta el objetivo. Este algoritmo se implemento en tres funciones, una que obtiene los datos necesarios para el procedimiento, la segunda que calcula la distancia desde el inicio al nodo actual y después del nodo actual al final y la otra que realiza el procedimiento.

Función `a_star()`**:**

Ésta es la función con la que se preparan las estructuras necesarias y se llama al procedimiento. Para llamar al procedimiento es necesario el llevar el punto inicial del algoritmo, el punto final al que se desea llegar y dos listas vacías, éstas para poder realizar un backtracking. Parámetros:

- ninguno.

Función `a_star_logic([F|R], [{Pi, Pj}|RC], {Si, Sj})`**:**

Esta función calcula la distancia del punto inicio al punto final, pasando por el punto donde me encuentro. Parámetros:

- `[F|R]` lista con la distancias de los vecinos al punto al final (heurísticas).
- `[{Pi, Pj}|RC]` lista de vecinos en mismo orden que `[F|R]`.
- `{Si, Sj}` punto de inicio para calculo de distancia del nodo al punto inicial.

Función `a_star_proc({Si, Sj}, {Fi, Fj}, {FringeCell, FringeHeu})`**:**

Ésta función es encargada de realizar la logística del procedimiento. Lo primero que hace es ver la posición en la que se encuentra, esto para verificar si ya se encuentra en la posición final. Si no se encuentra en la posición final solicita los vecinos, a los cuales se pueda mover, de la posición en la que se encuentra. Con estos vecinos, los agrega a las listas `FringeCell` y `FringeHeu` y después busca cual elemento del `FringeHeu` posee la menor distancia del inicio al final pasando a través del mismo para avanzar hacia esa posición. En caso de no obtener vecinos, revisa en las lista de `FringeCell` y `FringeHeu` para encontrar otra posibilidad de camino, en caso de que estas listas se encuentren vacías retorna que no es posible encontrar un camino valido. Parámetros:

- `{Si, Sj}` punto de inicio.
- `{Fi, Fj}` punto objetivo.
- `FringeCell` lista de puntos en el tablero a los cuales es posible visitarse.
- `FringeHeu` lista de las distancias del inicio al final a través del punto de los nodos en la lista `FringeCell` en su mismo orden.

2.3.3. Jump Points

Este algoritmos consiste en a partir de un nodo tomar una dirección y continuar por esta hasta encontrar el fin, no encontrar nada y salirse de la cuadricula o encontrar un *JumpPoint*. En caso de encontrar el final devuelve que encontró el objetivo, en caso de no encontrar nada, avisa que ese camino es inválido. En caso de encontrar *JumpPoint*, vuelve a realizar este algoritmo con dicho *JumpPoint*. En caso de encontrar un camino invalido, a partir del mismo nodo por el que llego a ese camino, selecciona un nuevo camino y continua por ese.

Función `jump_points()`:

Ésta función crea un nuevo proceso (`spawn`) que ejecutará el algoritmo JumpPoints. Parámetros:

- ninguno.

Función `jump_points_proc(OpenList, Parents, Gs, Fs, Opened, Closed, End = {Ex, Ey}, Board)`:

Función principal del proceso que intenta encontrar la ruta utilizando el algoritmo JumpPoints. Parámetros:

- `OpenList` lista de las coordenadas de los nodos JumpPoints para el presente nodo.
- `Parents` lista de los nodos padre para cada nodo en la OpenList.
- `Gs` lista de heurísticas asociada a cada nodo en la OpenList.
- `Fs` lista de distancias asociada a cada nodo en la OpenList.
- `Opened` lista con las coordenadas de los nodos JumpPoints.
- `Closed` lista con las coordenadas de los nodos ya visitados.
- `End` tupla con las coordenadas del nodo destino.
- `Board` vector que representa la cuadrícula sobre la cual se corre el algoritmo.

Función `identify_successors(Node, NodeG, Parent, Context, Payload = {Board, _})`:

Función que identifica el siguiente sucesor de un punto o nodo dado. Parámetros:

- `Node` nodo para buscar expandir.
- `NodeG` heurística del nodo.
- `Parent` tupla con las coordenadas del padre del nodo a expandir.
- `Context` tupla de 6 entradas con el contexto de la búsqueda {`OpenList, Parents, Gs, Fs, Opened, Closed`}. Ver la función `jump_points_proc(...)` para una descripción de cada uno de éstos.
- `Payload` una tupla con los datos estáticos requeridos por los algoritmos {`Board, End`}, la cuadrícu-

la y una tupla *End* con las coordenadas del nodo destino.

Función `find_neighbors({X, Y}, {Px, Py}, Board)`**:**

Ésta función es el *prunning* descrito en el artículo y permite identificar de todos los vecinos de un nodo cuales deben expandirse. Parámetros:

- `{X, Y}` tupla con las coordenadas del nodo actual.
- `{Px, Py}` tupla con las coordenadas del padre del nodo actual.
- `Board` vector que representa la cuadrícula del problema.

Función `jump({X, Y, Px, Py}, Payload = {Board, {Ex, Ey}})`**:**

Ésta función encuentra los puntos de salto o *JumpPoints* según se describe en el artículo. Parámetros:

- `{X, Y, Px, Px}` tupla con las coordenadas del nodo y su padre. Ver función anterior para una descripción más detallada de cada uno.
- `Payload` una tupla con los datos estáticos requeridos por los algoritmos `{Board, End}`, la cuadrícula y una tupla *End* con las coordenadas del nodo destino.

3. Análisis de resultados

El resumen de implementación de la presente tarea es el siguiente:

Algoritmo	**Implementado**
Greedy	Si
A*	Si
Jump points	Si

Cuadro 1: Perspectiva del producto por desarrollar.

Para las pruebas se utilizaron tres mapas generados por el equipo de trabajo. Dichos mapas de prueba pueden ser ubicados en el repositorio en:

```
/tests/test*.map
```

Los tres mapas son crecientes en dificultad. A continuación se incluye una captura de pantalla de cada algoritmo corriendo en cada uno de los mapas de prueba.

3.1. Ejecucion del programa

Para ejecutar el programa realice lo siguiente:

```
$ erl
1> c(search.erl).
2> board:start().
3> search:greedy().
4> search:a_star().
5> search:jump_points().
```

En el caso 2, y entre los pasos 3,4 y 5 se debe configurar el entorno y reiniciarlo en caso de que fuera requerido.

3.2. Mapa1

3.2.1. Greedy en mapa 1

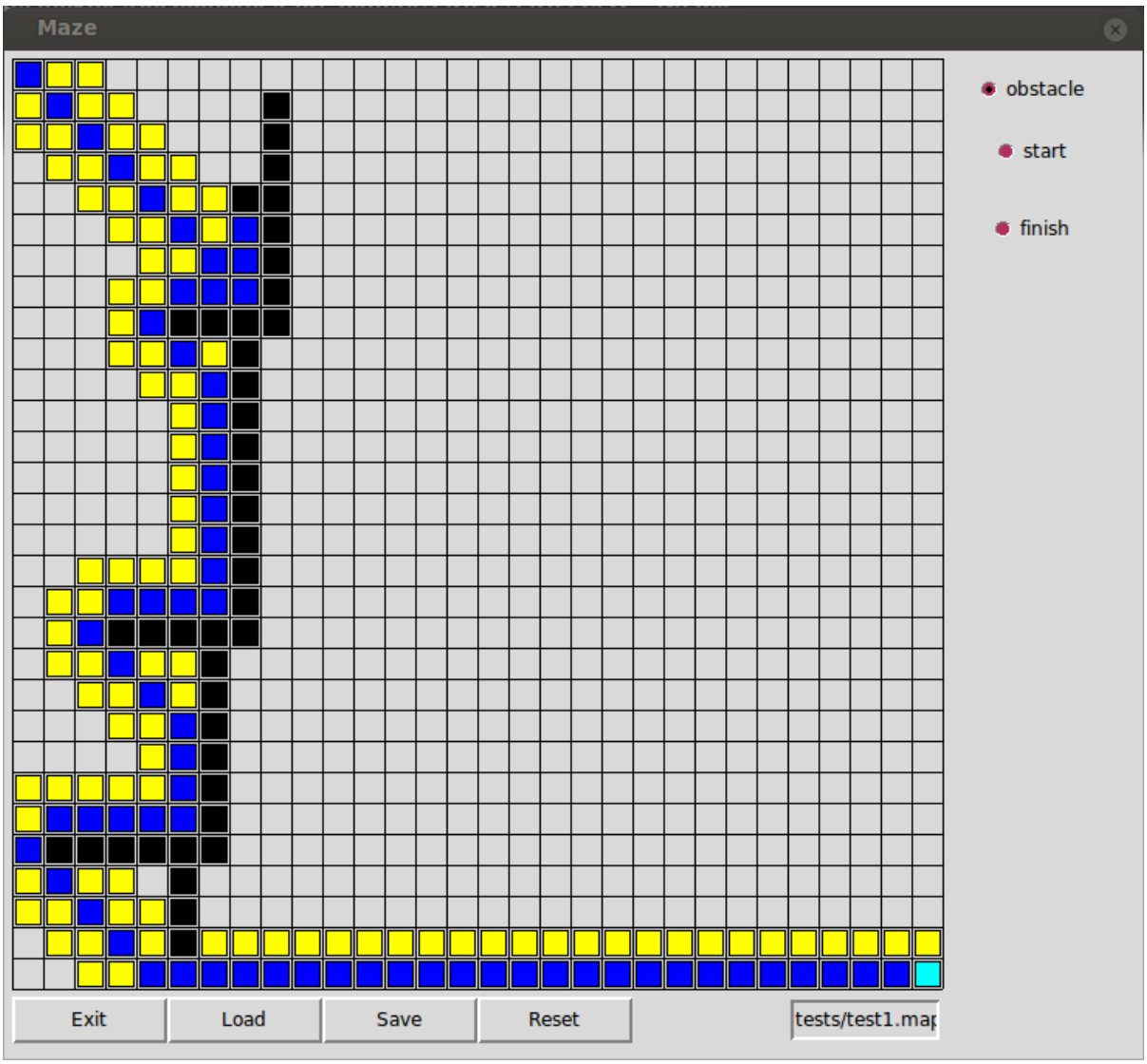

Figura 1: Ejecución del algoritmo greedy en el mapa de prueba 1.

3.2.2. A* en mapa 1

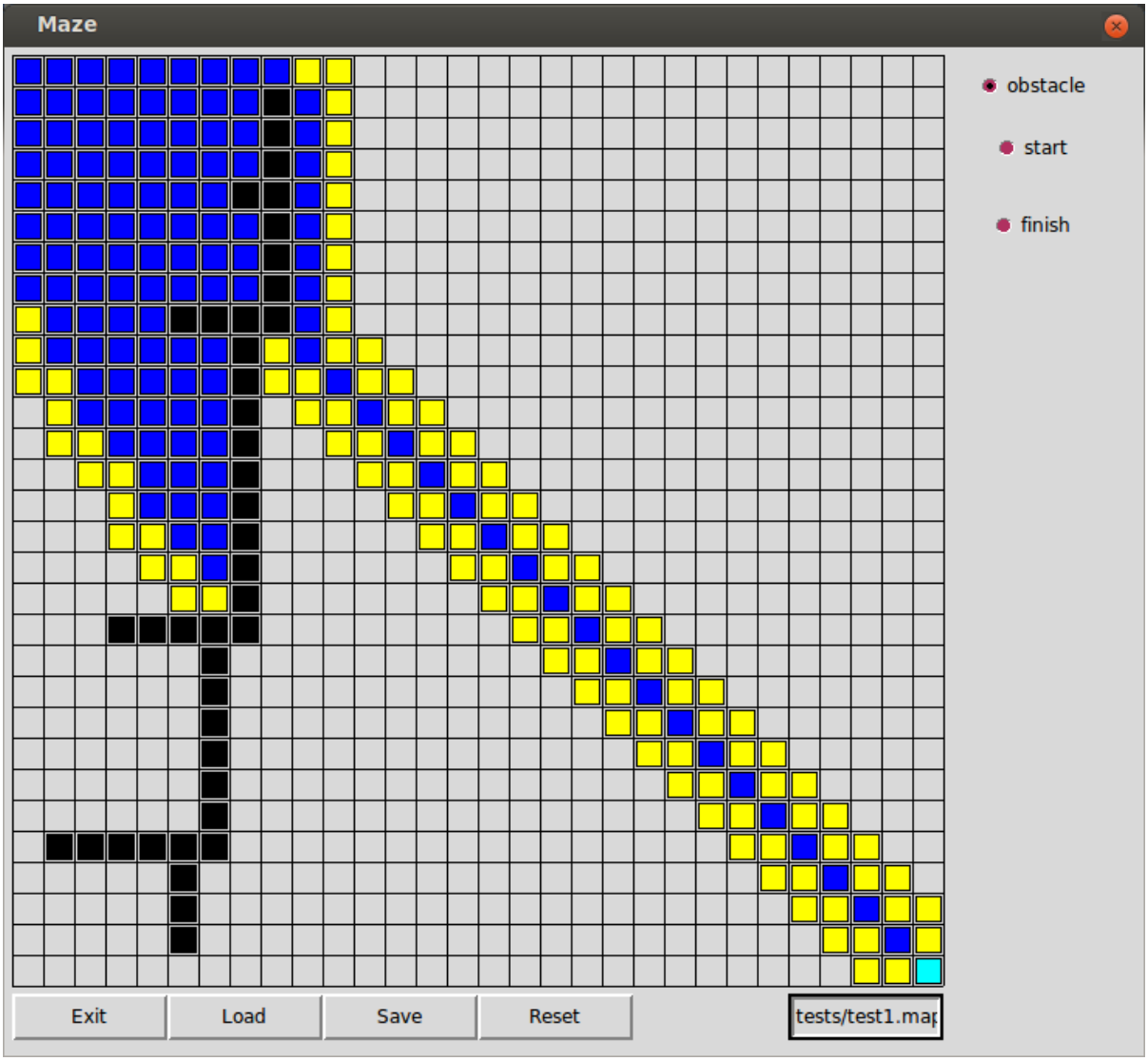

Figura 2: Ejecución del algoritmo A* en el mapa de prueba 1.

3.2.3. Jump points en mapa 1

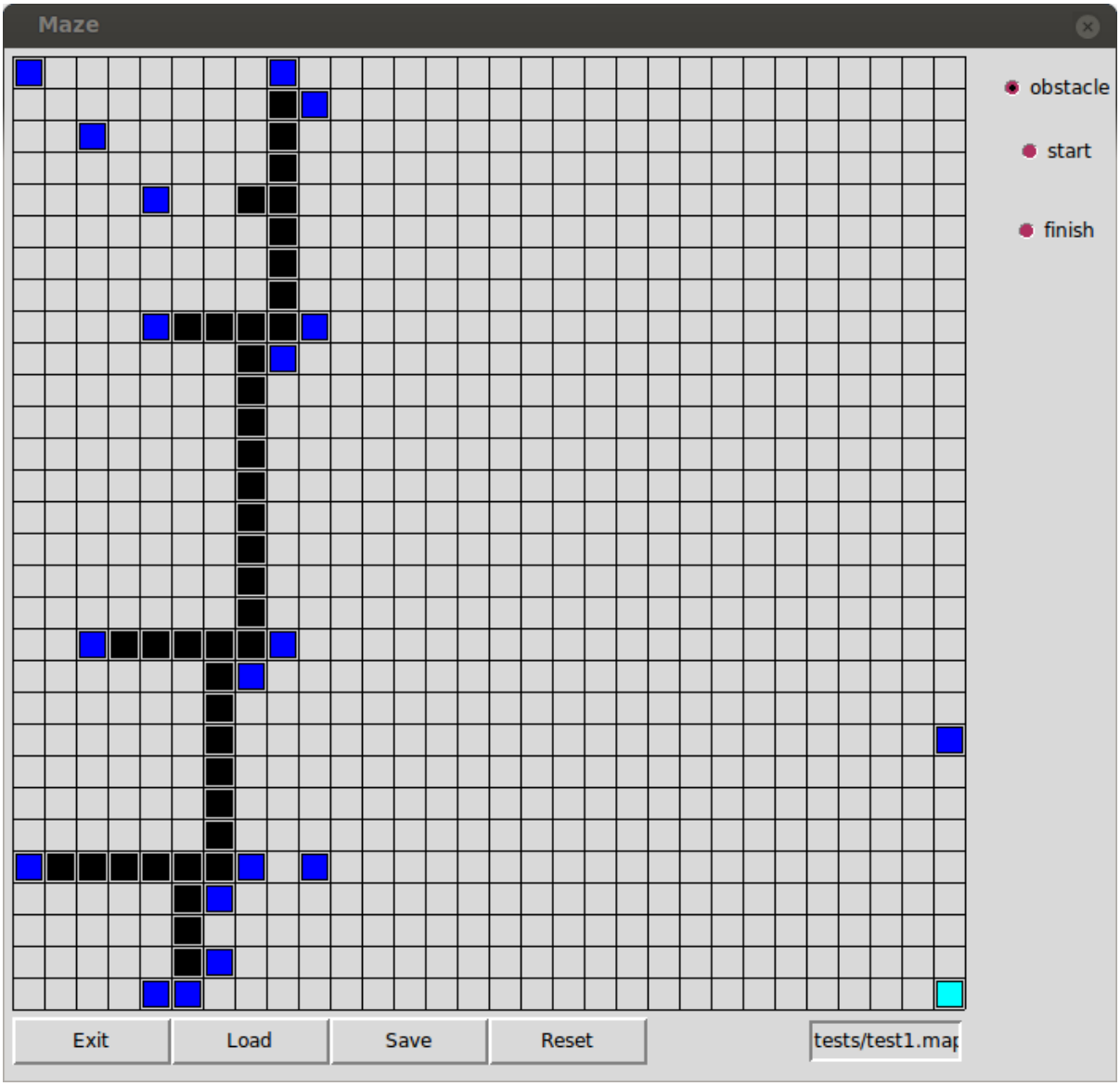

Figura 3: Ejecución del algoritmo Jump Points en el mapa de prueba 1.

3.3. Mapa2

3.3.1. Greedy en mapa 2

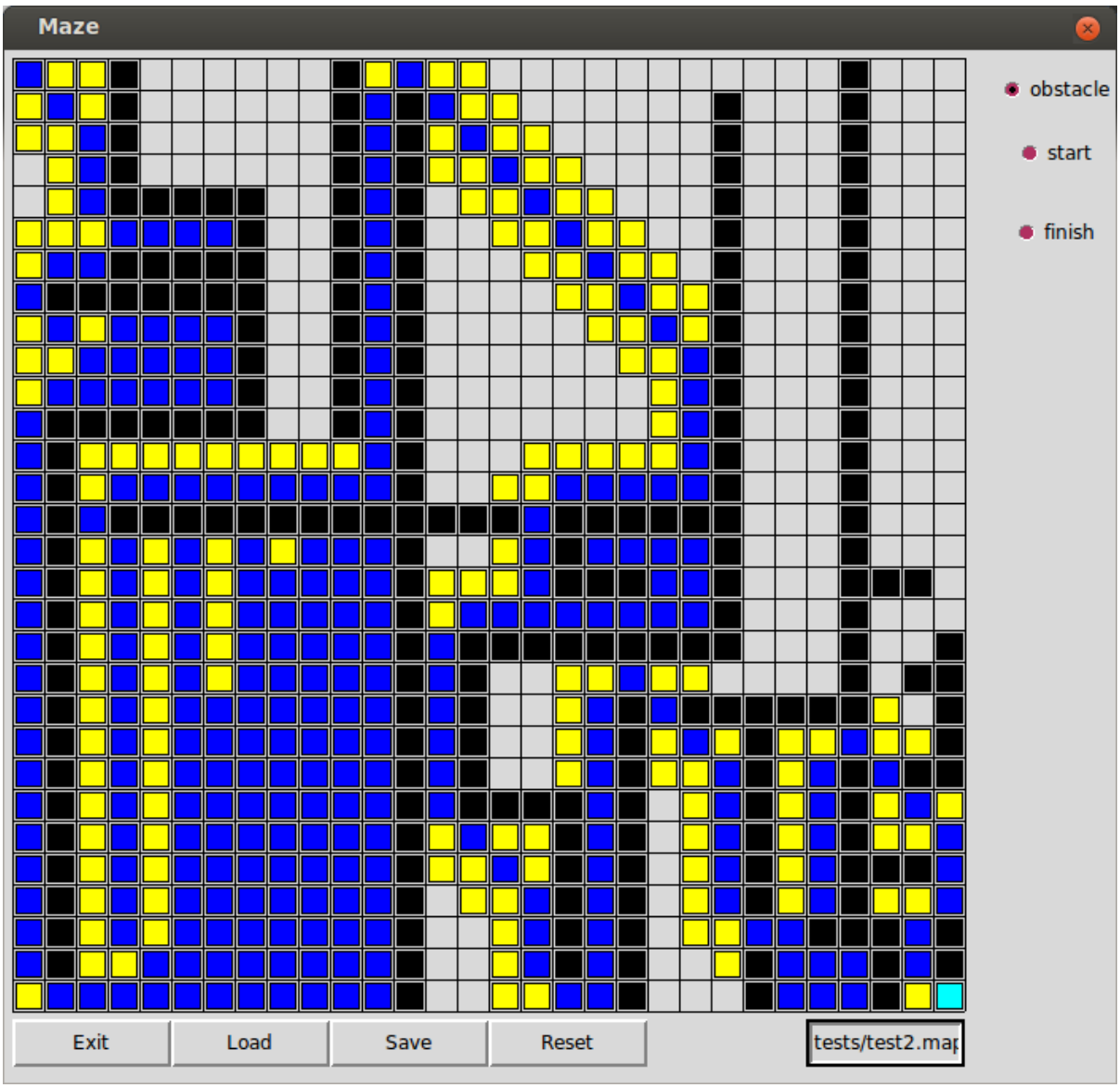

Figura 4: Ejecución del algoritmo greedy en el mapa de prueba 2.

3.3.2. A* en mapa 2

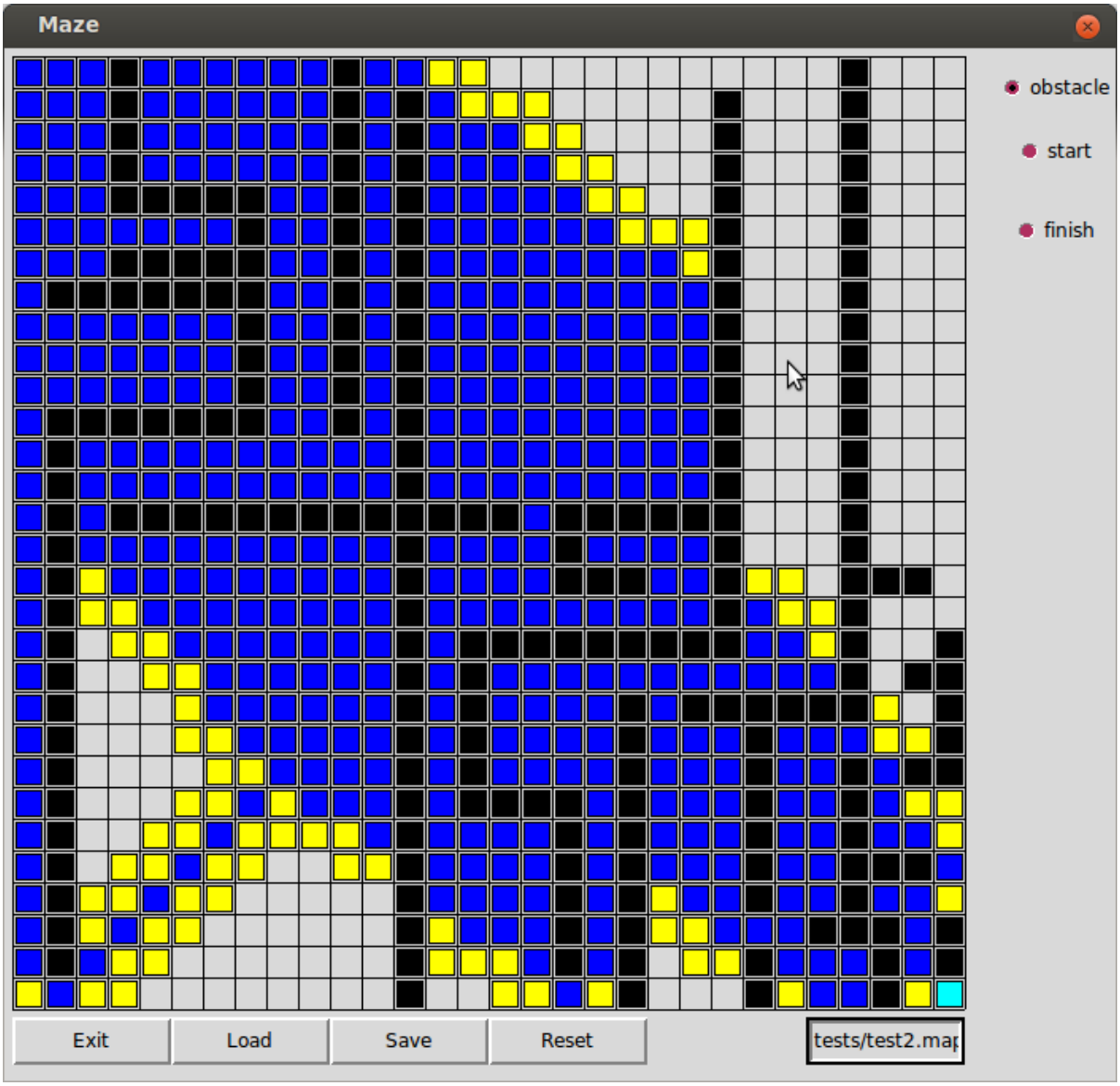

Figura 5: Ejecución del algoritmo A* en el mapa de prueba 2.

3.3.3. Jump points en mapa 2

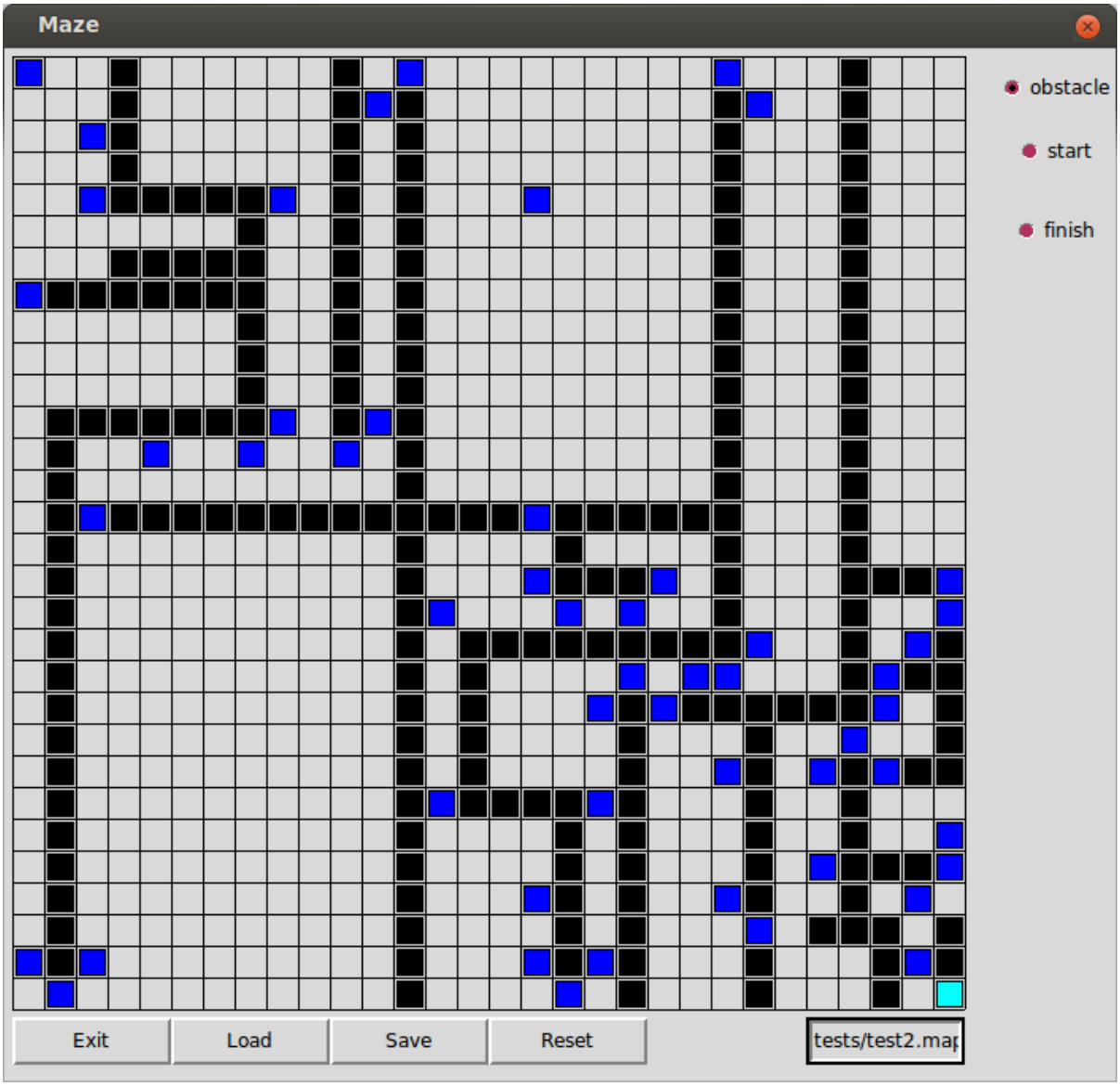

Figura 6: Ejecución del algoritmo Jump Points en el mapa de prueba 2.

3.4. Mapa3

3.4.1. Greedy en mapa 3

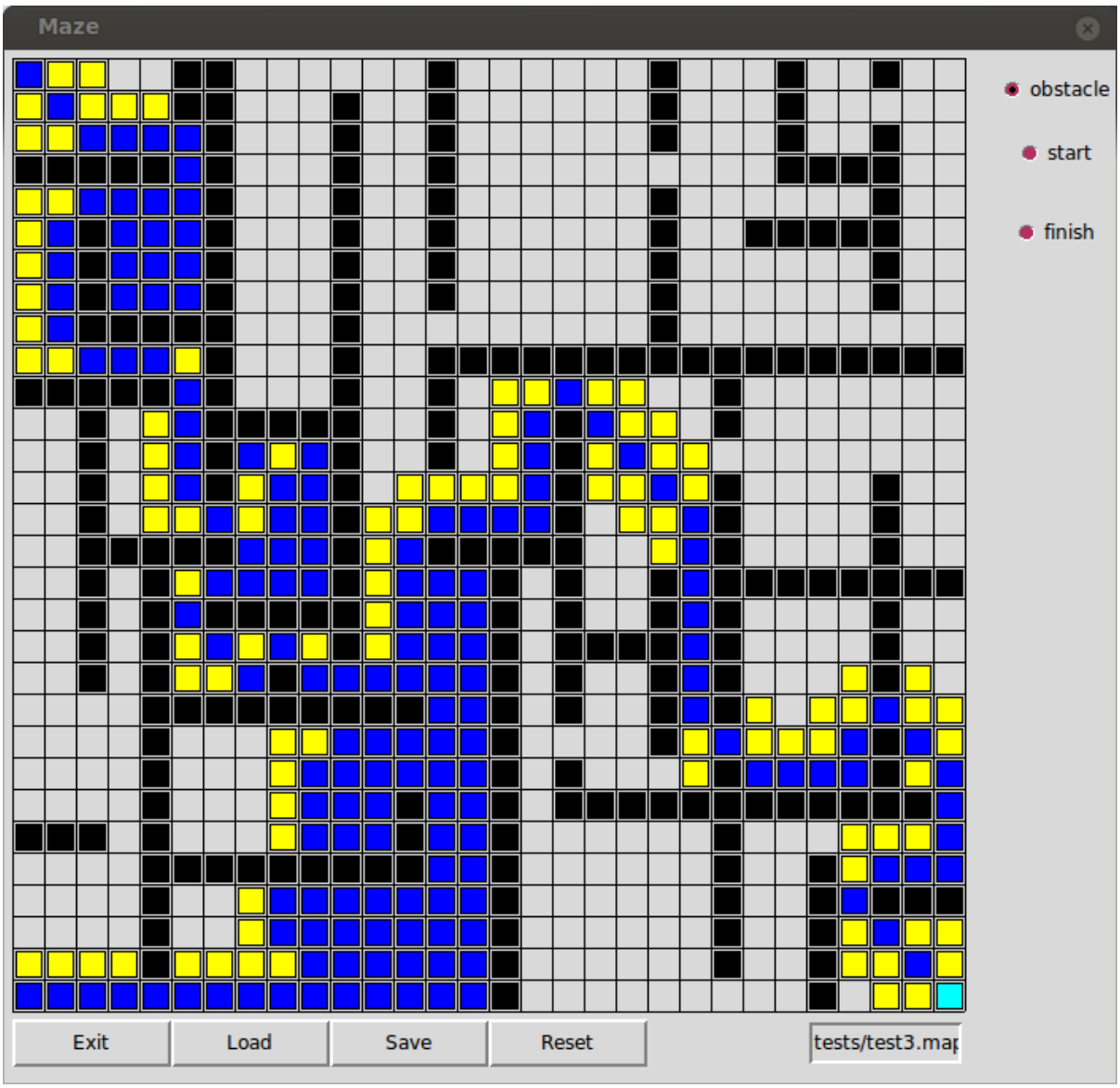

Figura 7: Ejecución del algoritmo greedy en el mapa de prueba 3.

3.4.2. A* en mapa 3

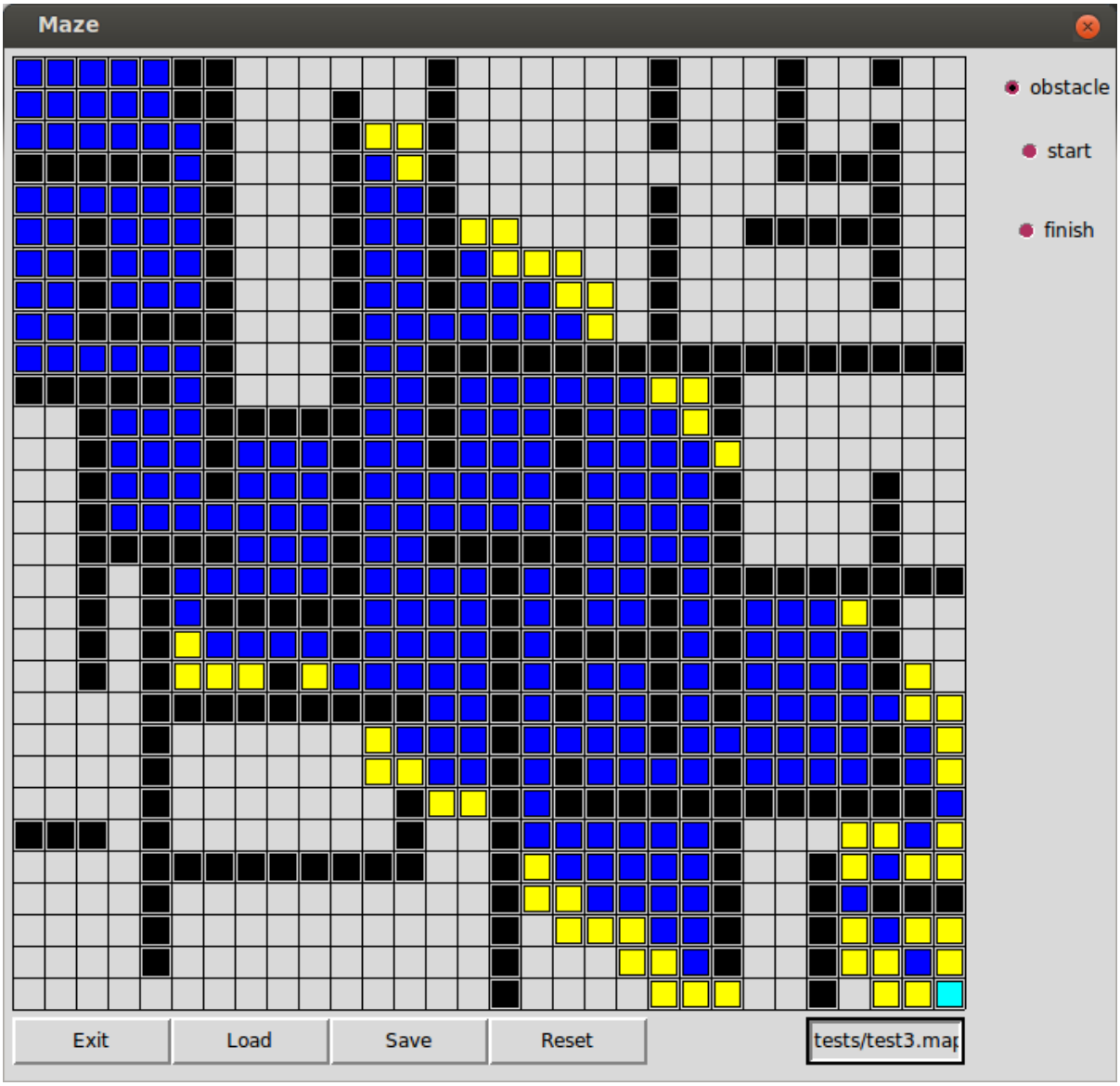

Figura 8: Ejecución del algoritmo A* en el mapa de prueba 3.

3.4.3. Jump points en mapa 3

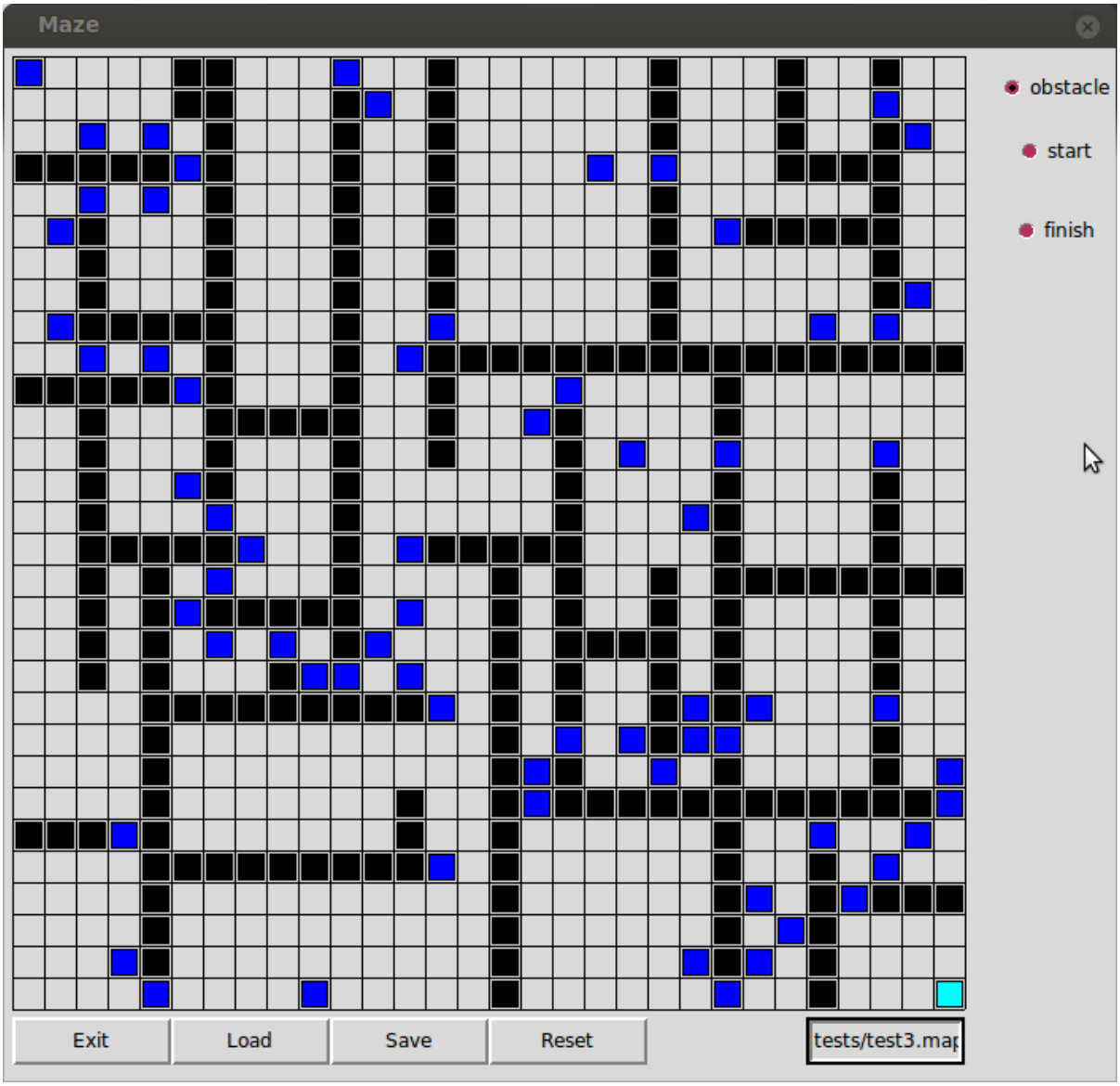

Figura 9: Ejecución del algoritmo Jump Points en el mapa de prueba 3.

4. Conclusiones

- El algoritmo Greedy es el segundo más rápido y es siempre encuentra una ruta, sin embargo, la ruta encontrada no es necesariamente la óptima. En la figura 1 del análisis de resultados se puede observar cómo el Greedy encuentra una ruta rápidamente pero ésta no es la óptima (la óptima sería irse por la apertura superior).
- El algoritmo Greedy necesita recordar todo el *Fringe* de la búsqueda para poder hacer el backtracking, lo que implica un *memory overhead* importante para problemas grandes.
- El algoritmo A* es el más lento de todos, sin embargo garantiza que la ruta encontrada sea la óptima. Como se puede observar en la figura 2, 5 y 8 el A* encuentra la ruta óptima pero al costo de una búsqueda más intensa.
- Al algoritmo A*, al igual que el Greedy tiene un *memory overhead* importante en problemas grandes pues también requiere recordar el *Fringe* de todo el recorrido.
- El algoritmo Jump Points es el más rápido y eficiente, sin embargo es el de mayor complejidad y requiere una lógica mucho más compleja. Al ser muy reciente (publicado en 2011) existen pocos artículos, implementaciones, tutoriales o explicaciones en relación a los dos anteriores, lo que dificulta la comprensión e implementación.
- El algoritmo Jump Points, a diferencia del Greedy y el A* no tiene un *memory overhead* importante pues no requiere recordar todo el *Fringe* y lo que requiere recordar es la lista de nodos de salto abiertos y los cerrados, lo cual es mínimo en comparación con la cantidad de nodos del *Fringe*.
- El algoritmo Jump Points al igual que el A* garantiza encontrar la ruta óptima y en menos tiempo (según el artículo hasta 10 veces mas rápido).
- Sin embargo, el algoritmo Jump Points sólo funciona en tableros (*grids*) uniformes de costo no variable entre saltos o casillas y bidireccionales, a diferencia del Greedy y del A* que funcionan en contexto de grafos dirigidos y de peso de arcos variable.

5. Referencias

- Harabor D. and Grastien A. (2011). *Online graph pruning for pathfinding on grid maps*. In Proceedings of the 25th National Conference on Artificial Intelligence (AAAI), San Francisco, USA.
- Xueqiao, X. (2011). *A comprehensive path-finding library in javascript*.
  Retrieved from https://github.com/qiao/PathFinding.js