Algoritmo de búsqueda A Estrella aplicado a resolución de laberintos

Gonzalez Valentín Armando Fabián

Universidad Tecnológica de la Mixteca

Instituto de Ingeniería en Computación

Resumen

En este documento se aborda una introducción sobre algoritmos de búsqueda en IA y la descripción del algoritmo A Estrella para la resolución de laberintos y cálculo de rutas más óptimas, se analiza el algoritmo y se muestra parte del código fuente al igual que capturas de pantalla de la implementación en lenguaje JavaScript.

Introducción

Las técnicas de búsqueda son una serie de esquemas de representación del conocimiento, que mediante diversos algoritmos nos permite resolver ciertos problemas desde el punto de vista de la I.A. Las técnicas de búsqueda se componen por un conjunto de estados: todas las configuraciones posibles en el dominio, estados iniciales: estados desde los que partimos, estados finales: las soluciones del problema, operadores: se aplican para pasar de un estado a otro. Algunas tecnicas de busqueda son:

• Búsqueda preferente por amplitud

• Búsqueda de costo uniforme

• Búsqueda preferente por profundidad

• Búsqueda limitada por profundidad

• Búsqueda por profundización iterativa

• Búsqueda direccional

Búsqueda heurística

• Búsqueda avara.

• Búsqueda A*.

Búsqueda limitada por la capacidad de la memoria:

- La búsqueda A* por profundización iterativa
- La búsqueda A* acotada por la memoria simplificada.

Planteamiento del problema y solución

Se plantea implementar un algoritmo que resuelve un laberinto de forma óptima. Para la solución de esta problemática se ha optado por el algoritmo A Estrella que sin duda destaca por su eficiencia y que además nos proporciona una ruta más óptima a la hora de una búsqueda. Esta implementación del algoritmo A Estrella funciona de forma similar al algoritmo recursivo Backtracker, lo que se hace es hacer uso de tres pilas o stacks para almacenar las posibles rutas e ir descartando las menos óptimas y así elegir la que tenga una mejor heurística.

Desarrollo e implementación

El algoritmo funciona de la siguiente manera: Se necesita una coordenada de partida y una de llegada, teniendo la coordenada de inicio se inicia con un análisis de las casillas vecinas en cuatro posiciones (izquierda, derecha, arriba, abajo) esto para verificar que están libres y que no se han visitado, mediante una función heurística se va calculando el costo de cada ruta(distancia a la meta). En una pila llamada **Openset** se van almacenando las casillas que forman parte de una posible ruta y que no hayan sido visitadas, en otra pila llamada **Closeset** se guardan los caminos que si conducen a la meta, una vez terminado el análisis de todo el laberinto se procede a

analizar el **Closeset** y se estima la ruta con menor coste gracias a la función heurística y de esta forma se obtiene la ruta más óptima para la meta. Cuando el algoritmo encuentra una pared retrocede y marca ese camino como no óptimo y busca otra ruta, en caso de que termine de visitar todas las casillas y no se encuentre ninguna ruta que conduzca a la meta se marca como no encontrado, la ruta más óptima se almacena en una pila llamada **camino** y se retorna el camino de vuelta desde la meta hasta el inicio.

Código

Heurística

La heurística es la estimación de coste de cada ruta, si bien en ocasiones no provee la más óptima provee la que mejor funciona de entre todas las soluciones de forma general en todos los algoritmos de búsqueda semejantes al A Estrella.

En la siguiente imagen se muestra la función que calcula los costes entre casillas.

```
//CALCULA LAS DISTANCIAS
function heuristica(a,b){
  var x = Math.abs(a.x - b.x);
  var y = Math.abs(a.y - b.y);
  var dist = x+y;
  return dist;
}
```

Imagen 1. Función heurística

En el siguiente extracto de código se muestra cómo se calculan los vecinos de cada casilla

```
//MÉTODO QUE CALCULA SUS VECNIOS
this.addVecinos = function(){
  if(this.x > 0)
    this.vecinos.push(escenario[this.y][this.x-1]);   //vecino izquierdo

  if(this.x < filas-1)
    this.vecinos.push(escenario[this.y][this.x+1]);   //vecino derecho

  if(this.y > 0)
    this.vecinos.push(escenario[this.y-1][this.x]);   //vecino de arriba

  if(this.y < columnas-1)
    this.vecinos.push(escenario[this.y+1][this.x]); //vecino de abajo
}
```

Imagen 2. Cálculo de casillas vecinas

En el apartado de **desarrollo e implementación** se hace una descripción del algoritmo, aquí solo se muestra el código del mismo.

```
function algoritmo(){
  //SEGUIMOS HASTA ENCONTRAR SOLUCIÓN
  if(terminado!=true){
    //SEGUIMOS SI HAY AlGO EN OPENSET
    if(openSet.length>0){
      var ganador = 0;  //índiCe o posición dentro del array openset
      //evaluamos que OpenSet tiene un menor coste / esfuerzo
      for(i=0; i<openSet.length; i++){
        if(openSet[i].f < openSet[ganador].f){
          ganador = i;
        }
      }
      //Analizamos la casilla ganadora
      var actual = openSet[ganador];
      //SI HEMOS LLEGADO AL FINAL BUSCAMOS EL CAMINO DE VUELTA
      if(actual === fin){
        var temporal = actual;
        camino.push(temporal);
        while(temporal.padre!=null){
          temporal = temporal.padre;
          camino.push(temporal);
        }
```

Imagen 3. Algoritmo

```
    console.log('camino encontrado');

    terminado = true;
  }
  //SI NO HEMOS LLEGADO AL FINAL, SEGUIMOS
  else{
    borraDelArray(openSet,actual);
    closedSet.push(actual);
    var vecinos = actual.vecinos;
    //RECORRO LOS VECINOS DE MI GANADOR
    for(i=0; i<vecinos.length; i++){
      var vecino = vecinos[i];
      //SI EL VECINO NO ESTÁ EN CLOSEDSET Y NO ES UNA PARED, HACEMOS LOS CÁLCULOS
      if(!closedSet.includes(vecino) && vecino.tipo!=1){
        var tempG = actual.g + 1;
        //si el vecino está en OpenSet y su peso es mayor
        if(openSet.includes(vecino)){
          if(tempG < vecino.g){
            vecino.g = tempG;     //camino más corto
          }
        }
```

Imagen 4. Algoritmo

```
lab.js > ...
                else{
                  vecino.g = tempG;
                  openSet.push(vecino);
                }
                //ACTUALIZAMOS VALORES
                vecino.h = heuristica(vecino,fin);
                vecino.f = vecino.g + vecino.h;
                //GUARDAMOS EL PADRE (DE DÓNDE VENIMOS)
                vecino.padre = actual;
              }
            }
          }
        }

        else{
          console.log('No hay un camino posible');
          terminado = true;   //el algoritmo ha terminado
        }
      }
    }
```

Imagen 5. Algoritmo

Algoritmo en ejecución

En pantalla se muestra de forma gráfica el funcionamiento del algoritmo para resolver el laberinto, se inicia haciendo la inspección de las casillas que conforman el laberinto coloreando de verde dichas casillas, las rutas que al final tienen una casilla en amarillo indican las que pudieron ser posibles rutas para la meta pero su coste era demasiado o no finalizaban a la meta(se muestran en rojo las cordenadas de dichas rutas posibles). El camino se colorea de color rojo y este es el más optimo de todos los posibles.

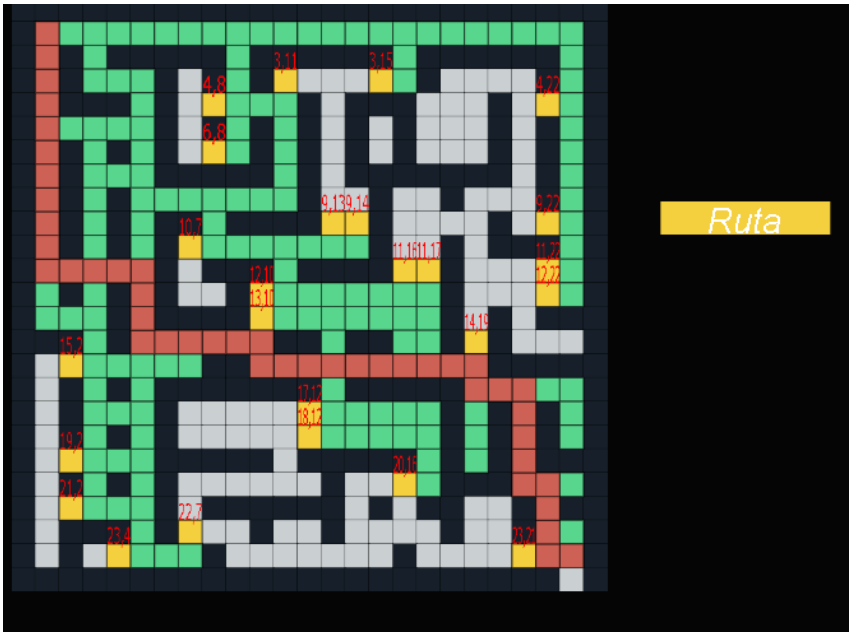

Imagen 6. Resolución de laberinto 1

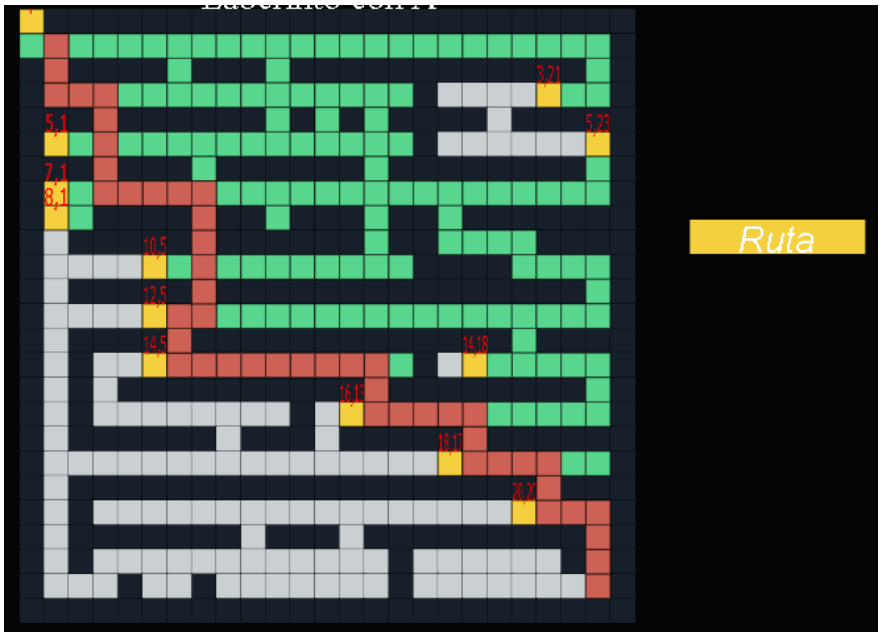

Imagen 7. Resolución de laberinto 2

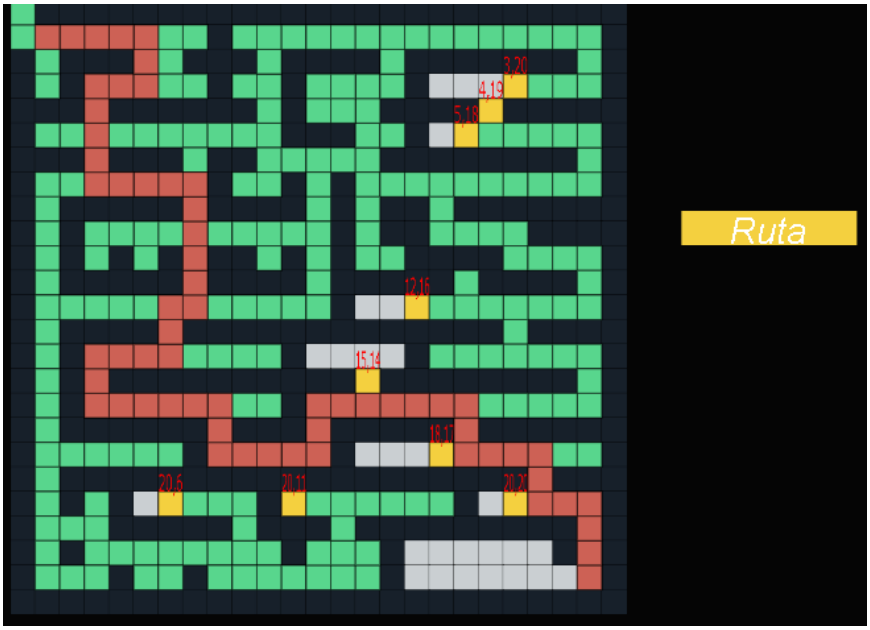

Imagen 8. Resolución de laberinto 3

Conclusión

El algoritmo A Estrella es usado en muchas aplicaciones en las que se requiere encontrar una ruta pero no solo eso, sino que se requiere encontrar la más óptima de entre todas las posibles. Una gran desventaja de este método es la memoria ya que es un algoritmo que tiene que analizar todas los nodos posibles, esto resulta sencillo si hablamos de un conjunto pequeño de nodos sin encambio si se tratase de un conjunto de cientos o miles de nodos esto implicaría un gran problema en el tiempo de ejecución. Existen variantes del algoritmo que resuelve esta problemática pero en este caso resulta eficaz para dar solución a un laberinto pequeño, otros algoritmos de búsqueda exhaustiva que tienen un funcionamiento similar son el algoritmo de búsqueda en anchura y en profundidad en los cuales no se puede avanzar a inspeccionar el siguiente nivel sin haber concluido o completado el nivel anterior del árbol de búsqueda. Otro algoritmo que suele ser muy utilizado para dar solución a este tipo de problemas es el algoritmo recursivo Backtracker el cual genera una copia de cada casilla de forma recursiva hasta encontrar una ruta a la meta, claro que se tiene que ser más cuidadoso a la hora de usar este algoritmo ya que genera más restricciones para validar una ruta a diferencia de un algoritmo iterativo pues se puede dar el caso de generar rutas repetidas o de visitar de forma infinita la misma casilla.

Existen gran variedad de algoritmos que resulven laberintos cada uno con sus pros y contras pero el objetivo es dar pauta a la implementación de dichos algoritmos y su aplicación en la vida real.

Referencias

https://www.javatpoint.com/search-algorithms-in-ai#:~:text=In%20Artificial%20Intelligence%2C%20Search%20techniques,agents%20and%20use%20atomic%20representation.

https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/

https://towardsdatascience.com/ai-search-algorithms-every-data-scientist-should-know-ed0968a43a7a