Grado en Ingeniería informática

Sistemas Inteligentes

Práctica 1.Búsqueda

Documentación del proyecto

ÍNDICE

Introducción

Esta memoria pertenece a la práctica 1; búsqueda; de la asignatura Sistemas inteligentes del grado en ingeniería informática de la universidad de Alicante.

A lo largo de esta memoria se explicará en que consiste el algoritmo de búsqueda A*y se detallará la implementación del mismo. En este caso el algoritmo se implementa para definir el comportamiento de los personajes del clásico juego el comecocos o pacman. Por tanto existirá la implementación del algoritmo para los fantasmas así como el propio pacman. Además se realizará la traza de un problema, se abordarán las distintas heurísticas implementadas y se tratarán diferentes casuísticas.

El algoritmo A*

El algoritmo A* o A estrella consiste en un algoritmo de búsqueda que permite encontrar el camino de menor coste entre dos puntos. Este algoritmo de búsqueda es completo; si existe una solución siempre dará con ella; y óptimamente eficiente, ya que en comparación al resto de algoritmos de búsqueda es el que expande menos nodos antes de encontrar la solución.

Es un algoritmo de búsqueda informado, utiliza una heurística para ordenar los nodos aún por explorar, por ello que sea un algoritmo óptimo.

Implementación de A* para los fantasmas

Como ya se ha mencionado se aborda la implementación para el comportamiento de los personajes de pacman. En el caso de los fantasmas su comportamiento debe de ser el de perseguir a pacman con el objetivo de atraparlo. Para ello deberán de desplazarse en aquella dirección en la que se obtenga el recorrido de menor coste hasta pacman. Luego se debe de conocer antes de desplazarse cuál es el camino de menor de coste hasta pacman y tomar la dirección del primer movimiento.

Hacer mención que los personajes solo pueden moverse arriba, abajo, izquierda y derecha.

Para poder hacer posible la implementación de este algoritmo se han tenido que crear dos clases adicionales.

En primer lugar la clase Nodo, esta clase almacenará el una posición en el tablero así como el coste de desplazarse hasta la misma, un valor heurístico entre esa posición y pacman y cuál es el Nodo antecesor al mismo.

En este caso la clase nodo se compone de las siguientes variables:

```
int h;//heuristica
int g;//coste del movimiento
int n;//columna
int m;//fila
Nodo padre;//corresponde al nodo padre
int movimiento;//indica el movimiento que se ha realizado para llegar a esa posición
```

Además de las variables la clase nodo tiene diferentes métodos que facilitan el uso de esta clase en la implementación.

Para evitar alargar en exceso esta memoria solo se mencionarán los más significativos, ya que el código está debidamente comentado facilitando la comprensión del mismo.

A continuación se muestra un método de la clase:

```
//devuelve un nodo correspondiente a desplazarse hacia arriba
Nodo arriba(){
    Nodo nodo = new Nodo();
    nodo.h = 0;
    nodo.g = this.g+1;//nos hemos desplazado
    nodo.n = this.n;
    nodo.m  = this.m-1;
    nodo.padre = new Nodo();
    nodo.padre.asigna(this);
    nodo.movimiento = Laberinto.ARRIBA;
    return nodo;
}
```

Como vemos en la imagen anterior, este método devuelve un nodo hijo correspondiente a haberse desplazado hacia arriba, para ello:

- Crea un nuevo nodo
- Aumenta en una unidad el coste del nodo, g.
- Asigna la coordenada correspondiente a las filas, decrementada en una unidad.
- Asigna el nodo actual al padre del nuevo nodo.
- Y asigna el movimiento arriba a la variable movimiento del nuevo nodo
- Retorna el nuevo nodo creado

Existen otros tres métodos correspondientes al resto de movimientos.

También se ha implementado el método iguales(Nodo), este método retorna un booleano en función de si dos nodos son iguales o no lo son. Teniendo en cuenta que dos nodos se considerarán iguales cuando ambos ocupen la misma coordenada. Este método además de recibir por parámetro un nodo puede recibir dos enteros, correspondientes a las columnas y las filas sucesivamente. El método es el siguiente:

```
//dos nodos serám iguales si tienen las mismas coordenadas
public boolean iguales(Nodo otro) {
    return otro.n == this.n && otro.m == this.m;
}
public boolean iguales(int cn, int cm){
    return cn == this.n && cm == this.m;
}
```

Para la función de evaluación del nodo; f = g + h; se ha implementado el método f() que retorna el valor de dicha suma, el método es el siguiente:

```
//método que devuelve el valor de f
public int f(){
    return g + h;
}
```

Por último el método calcH(int, int) lo veremos más adelante, la llamada de este método asigna a h un valor heurístico de la distancia entre la posición del nodo y las coordenadas pasadas por parámetro.

Además de la clase nodo también se ha implementado la clase ListaNodos(), la cual define el comportamiento de un arrayList de nodos.

Por lo que solo tiene una variable llamada lista la cual se define como un arrayList de nodos, de la siguiente forma:

```
private ArrayList<Nodo> lista = new ArrayList<Nodo>();
```

Es una variable privada, solo se va a acceder a este desde la propia clase.

Como métodos que destacar están los siguientes:

```
//retorna un booleano en función de si el nodo esta en la lista o no
boolean contiene(Nodo n){
    Iterator<Nodo> it = lista.iterator();
    while(it.hasNext()){
        if(n.iguales(it.next())){
            return true;//encontrado
        }
    }
    return false;//no encontrado
}
```

El método contiene es equivalente al método contains de un lista convencional, solo que en este caso utiliza el método iguales, por lo que se entiende que contendrá el nodo con tan solo coincidir en coordenadas con el nodo pasado como parámetro.

Este método recorre la lista por medio de un iterador retornando true cuando se cumpla la condición o en caso contrario false si recorre toda la lista sin éxito.

```
//retorna el nodo de menor f en la lista
Nodo menor(){
    Nodo menor = new Nodo();
    Nodo nodoAux = new Nodo();
    Iterator<Nodo> it = lista.iterator();
    menor.asigna(it.next());
    while(it.hasNext()){
        nodoAux = it.next();
        if(nodoAux.f() < menor.f()){
            menor.asigna(nodoAux);
        }
    }
    return menor;
}
```

Este otro método retornar el nodo con menor función de evaluación almacenado en la lista, para ello con un iterador se recorre la lista almacenando el nodo con menor f en menor, de tal manera que una vez recorrida toda la lista menor contiene el nodo con una menor función de evaluación.

No se han mencionado la totalidad de métodos implementados para esta clase, el resto de métodos están debidamente identificados en el código.

Vistas las clases anteriores podemos proceder a la implementación del algoritmo:

```
ListaNodos listaInterior = new ListaNodos();
ListaNodos listaFrontera = new ListaNodos();
Nodo inicial = new Nodo();
inicial.n = laberinto.obtenerPosicionFantasma(numeroFantasma)[0];
inicial.m = laberinto.obtenerPosicionFantasma(numeroFantasma)[1];
inicial.calcH(laberinto.obtenerPosicionPacman()[0],laberinto.obtenerPosicionPacman()[1]);
listaFrontera.add(inicial);
Nodo nodoAux = new Nodo();//nodo auxiliar
```

En esta primera parte del algoritmo se inicializan dos variables conocidas como listaInterior y listaFrontera las cuales son dos ListaNodos; la primera lista guardará los nodos expandidos mientras que la segunda los nodos aún por expandir.

A lista frontera se le añade el nodo inicial, este nodo se inicializa con las coordenadas del fantasma y se calcula el valor de su heurística antes de ser añadido a la lista.

También se crea un nodo conocido como nodoAux, este nodo se utilizará a lo largo del algoritmo.

```
while(!listaFrontera.isEmpty()){
```

Comienza un bucle while, este bucle iterará mientras la listaFrontera no este vacía, si deja de iterar a causa de que la lista este vacía querría decir que no existe un camino que resuelva el problema.

Dentro del bucle sucede lo descrito a continuación.

```
nodoAux.asigna(listaFrontera.menor());
listaFrontera.remove(nodoAux);
listaInterior.add(nodoAux);
```

Se obtiene el nodo de menor f y se asigna al nodo auxiliar, se borra de la listaFrontera y se añade a la ListaInterior. El nodo de menor f significa que es el nodo más prometedor ya que la suma del coste que le ha llevado a esa posición más la heurística es menor a la del resto de nodos. Es decir, probablemente ese nodo sea el que tiene un menor camino hasta el objetivo, lo que optimiza bastante el algoritmo.

```
if(nodoAux.iguales(laberinto.obtenerPosicionPacman()[0]
                   ,laberinto.obtenerPosicionPacman()[1])){//meta
    encontrado = true;
    System.out.println("ENCONTRADO");
    coste_total = (double) nodoAux.g;
    expandidos = listaInterior.size();
    int paso = nodoAux.g;
    //encontrar el nodo inicial y saber que movimiento se ha realizado
    while(nodoAux.padre.padre != null){//el objetivo es encontrar el inicial
        camino[nodoAux.m][nodoAux.n] = 'x';
        camino_expandido[nodoAux.m][nodoAux.n] = paso;
        paso--;
        nodoAux.asigna(nodoAux.padre);
    }
    result = nodoAux.movimiento;
    camino[nodoAux.m][nodoAux.n] = 'x';
    camino_expandido[nodoAux.m][nodoAux.n] = paso;
    paso--;
    nodoAux.asigna(nodoAux.padre);
    camino[nodoAux.m][nodoAux.n] = 'x';
    camino_expandido[nodoAux.m][nodoAux.n] = paso;
    break;//para salir del bucle
}
```

Este condicional se cumplirá cuando el nodo llegue a la meta; es decir; el nodo tenga las mismas coordenadas que tiene pacman. Por lo que si el nodo está en la meta significa que se ha encontrado el camino de menor coste entre el fantasma y pacman.

Luego ahora debemos de obtener cual ha sido el primer movimiento realizado para completar ese camino óptimo. Para ello el bucle while que iterará hasta encotrar un nodo cuyo padre tenga un padre nuelo, es decir el nodo hijo del nodo inicial, y el movimiento que nos ha llevado a ese nodo es el movimiento a retornar. Por ello result es igual a nodoAux.movimiento. Dado que se ha encontrado la solución no será necesario que el primer While siga iterando, por eso hay un break.

El resto de líneas permiten dar valor a las variables que más adelanta mostrarán la salida por pantalla pedida en el enunciado.

De no haberse encontrado la solución el bucle while seguiría con la iteración tal y como sigue el código a continuación.

```
//se crea un array de nodos con los 4 hijos del nodo actual
Nodo hijos[] = new Nodo[4];
hijos[0] = nodoAux.arriba();
hijos[1] = nodoAux.abajo();
hijos[2] = nodoAux.derecha();
hijos[3] = nodoAux.izquierda();
```

En este trozo de código se ha rellenado un vector de nodos con los nodos correspondientes a haberse desplazado en todas las direcciones.

```
for(int i = 0; i < 4;i++){
    if(!listaInterior.contiene(hijos[i])
        && laberinto.obtenerPosicion(hijos[i].n,hijos[i].m) != 1){
        if(!listaFrontera.contiene(hijos[i])){
            hijos[i].calcH(laberinto.obtenerPosicionPacman()[0],
                           laberinto.obtenerPosicionPacman()[1]);
            listaFrontera.add(hijos[i]);
        }else if(hijos[i].g < listaFrontera.getG(hijos[i])){
            //actualizar
            listaFrontera.remove(hijos[i]);
            hijos[i].calcH(laberinto.obtenerPosicionPacman()[0],
                           laberinto.obtenerPosicionPacman()[1]);
            listaFrontera.add(hijos[i]);
        }
    }
}
```

En esta parte se recorre el vector anteriormente rellenado con todos los hijos, el primer condicional será cierto si, el hijo no esta en listaInterior y si es una posición accesible; es decir; no es un muro. De no cumplirse este nodo se descarta, si si se cumple se pasa al siguiente condicional.

Si el hijo no esta en listaFrontera, se calcula su heurística y se añade a la listaFrontera. Si el hijo esta en listaFrontera se pasa a evaluar si este nuevo camino hasta esa posición ha sido más corto de la manera en que se ha llegado anteriormente; de esta manera se evitan bucles y caminos que no sean el de menor coste; para ello se compara el coste del hijo actual frente al coste del nodo ya almacenado, en caso de que el coste sea menor para el hijo actual se elimina el nodo anterior y se actualiza la lista frontera con el nodo que llega a esa posición de una manera más óptima.

Implementación de A* para pacman

Ahora abordaremos la implementación de A estrella para el comportamiento de pacman. En este caso pacman debe de huir de los fantasma, para ello debe moverse a aquella posición en la que el camino de menor coste para el fantasma que le pueda atrapar tenga el mayor coste posible; es decir; que este lo más lejos posible del fantasma más cercano.

Para ello pacman debe tener en cuenta de cada posible movimiento, el coste que tendrían los fantasma en atraparla estando en esa posición y quedarse el coste menor para atraparle en cada posición, y de esos costes elegir el mayor y averiguar a que movimiento pertenece para posteriormente desplazarse hacia esa posición.

Para hacer posible la implementación del comportamiento de pacman, se ha creado la siguiente clase:

```
//clase que define un posible movimiento de pacman
class movimientoPacman{

    int n;//columas
    int m;//filas
    int movimiento;//almacena el movimiento que se ha llevado a cabo para llegar a esa posición
    int coste;//coste del camino del fantasma mas cercano
```

Esta clase tiene varios métodos que de nuevo evitaremos mencionar, pasando a mencionar los más significativos:

```
//crea un movimiento correspopndiente a moverse hacia arriba
void arriba(int n, int m){
    this.n = n;
    this.m = m-1;
    this.movimiento = Laberinto.ARRIBA;
    this.coste = 0;//habrá que calcular el coste
}
```

Este método es similar al de la clase nodo solo que en este caso para la clase movimientoPacman, se crea un movimiento correspondiente a desplazarse a una de las posiciones tomando como parámetros la posición actual de pacman, existen otros tres métodos para definir la totalidad de los movimientos disponibles.

Esta clase implementa el algoritmo A estrella, es el igual al A estrella definido anteriormente salvo que en lugar de retorna el primer movimiento realizado, retorna el coste del camino hasta el objetivo. A continuación se muestra la parte del código que varía:

```
if(nodoAux.iguales(nFantasma,mFantasma)){//es meta
    return nodoAux.g;
}
```

Como vemos en caso de que un nodo sea meta; es decir; en caso de que un nodo ocupe la posición objetivo, se retorna el coste de ese nodo. Un nodo ocupará la

posición objetivo cuando sus coordenadas coincidas con las del fantasma pasadas como parámetro, ya que estamos calculando el camino de menor coste desde el movimiento de pacman hasta el fantasma. Cabe mencionar que este método también recibe como parámetro el laberinto, de tal manera que podrá descartar aquellos nodos que correspondan a posiciones incorrectas.

Vista la clase podemos pasar a la implementación del método A estrella para pacman, este método debe de retornar el mejor movimiento de huida. Se muestra a continuación.

```
movimientoPacman movimientos[] = new movimientoPacman[4];
for(int i = 0; i < 4; i++){
    movimientos[i] = new movimientoPacman();
}
movimientos[0].arriba(laberinto.obtenerPosicionPacman()[0],
                      laberinto.obtenerPosicionPacman()[1]);
movimientos[1].abajo(laberinto.obtenerPosicionPacman()[0],
                      laberinto.obtenerPosicionPacman()[1]);
movimientos[2].derecha(laberinto.obtenerPosicionPacman()[0],
                      laberinto.obtenerPosicionPacman()[1]);
movimientos[3].izquierda(laberinto.obtenerPosicionPacman()[0],
                        laberinto.obtenerPosicionPacman()[1]);
movimientoPacman the_best = new movimientoPacman();
```

El código comienza creando e inicializando un vector de 4 movimientos, después rellena este mismo vector con los movimientos correspondientes a las cuatro posiciones a las que se puede desplazar pacman. Y luego crea un nuevo movimiento llamado the_best, este movimiento almacenará el mejor movimiento posible para pacman.

```
for(int i = 0; i < 4; i++){//para cada movimiento
    if(laberinto.obtenerPosicion(movimientos[i].n, movimientos[i].m) != 1){//que sea accesible
        movimientos[i].coste = movimientos[i].Aestrella(laberinto.obtenerPosicionFantasma(1)[0],
                                                        laberinto.obtenerPosicionFantasma(1)[1],
                                                        laberinto);//calculamos el coste para el primer fantasma
        for(int j = 2; j <= laberinto.numFantasmas; j++){//para cada fantasma menos el primero
            int costeNuevo = movimientos[i].Aestrella(laberinto.obtenerPosicionFantasma(j)[0],
                                                      laberinto.obtenerPosicionFantasma(j)[1],
                                                      laberinto);
            if(costeNuevo < movimientos[i].coste){//se asignará el menor coste posible
                movimientos[i].coste = costeNuevo;
            }
        }
        if(i == 0 || the_best.coste < movimientos[i].coste){
        //si es la primera iteración o el coste del nuevo movimiento es mayor que el mejor coste
            the_best.asignar(movimientos[i]);
        }
    }
}
result = the_best.movimiento;
return result;
```

La parte principal de esta implementación consiste en dos bucles anidados, uno de ellos nos permitirá recorrer los cuatro posibles movimientos y el otro en el caso de haber más de un fantasma calculará el coste para cada uno de ellos.

Hablemos del primer bucle for, este bucle recorre el vector de movimientos tomando cada posible movimiento de pacman y en primer lugar realiza un condicional para descartar aquellos movimientos a posiciones no válidas; al muro; de ser un movimiento válido calcula el coste al primer fantasma utilizando el algoritmo de búsqueda A* llamando al método que hemos descrito antes.

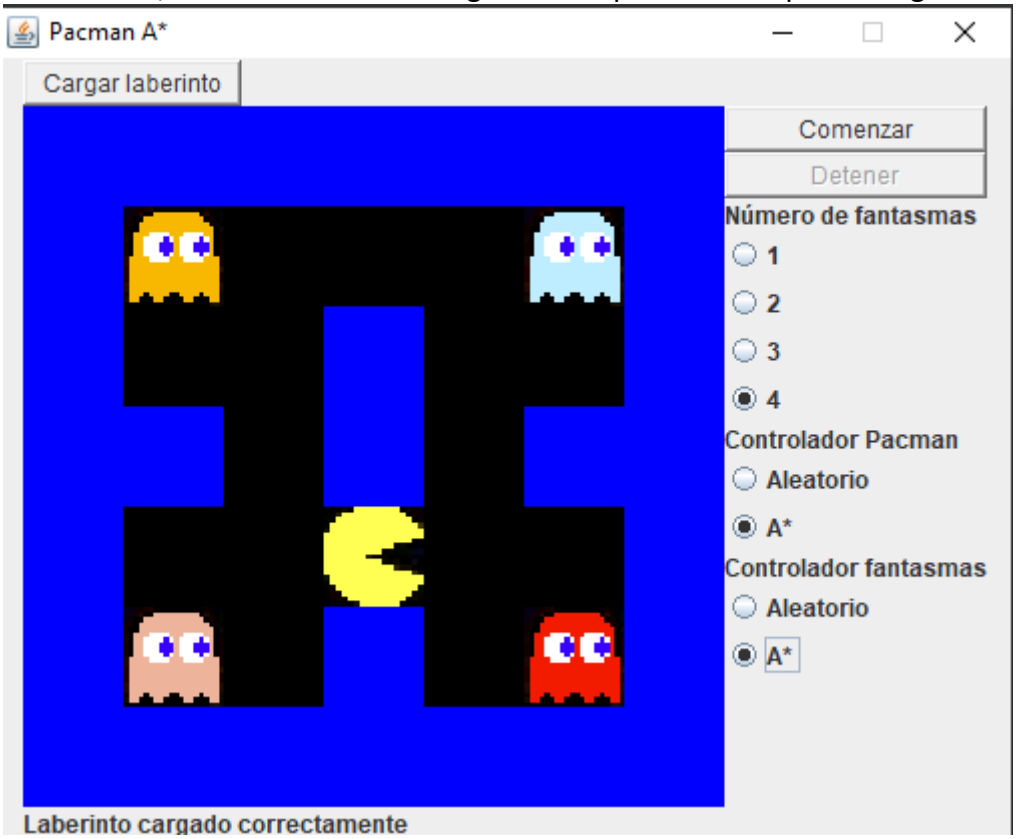
En el caso que el mapa tenga más de un fantasma, entonces se entrará en el segundo bucle for el cual calcula el coste del camino a cada fantasma y si este nuevo coste calcula es menor que el coste que ya tiene asignado el movimiento, entonces asigna ese nuevo coste. De tal manera que tras este bucle el movimiento tiene asignado el menor coste de entre todos los costes a todos los fantasmas.

Antes de terminar el primer bucle for, comprueba si se trata de la primera iteración; i = 0; y de ser así al movimiento the_best se le asigna el movimiento actual. De no ser la primera iteración entonces para ser asignado a the_best, el coste almacenado en ese movimiento debe de ser mayor que el coste ya asignado a the_best. De tal manera que al terminar este bucle the_best corresponde al movimiento de huida para pacman.

Para terminar el algoritmo, se devuelve the_best.movimiento que corresponde al movimiento que debe de realizar pacman.

Traza de un problema

Para ver aún más en detalle la implementación del algoritmo, a continuación se va a detallar la traza de un problema.
Para estas trazas, nos basaremos en el siguiente mapa 5x5. El mapa es el siguiente:



-para los fantasmas

Dado que la implementación es la misma para todos los fantasmas esta traza la realizaremos para un solo fantasma.
En primer lugar se crea el nodo inicial, correspondiente a las coordenadas actuales del fantasma. Ese nodo se añade a listaFrontera.
El código entra en el while, nodoAux toma el valor de este nodo inicial y dado que no es meta, se expande creando cuatro hijos.

Cada uno de estos hijos corresponde a haberse desplazado arriba, abajo, izquierda y derecha. Se recorrerán cada uno de estos hijos, se descartarán los nodos correspondientes a moverse arriba y a la izquierda, ya que estos nodos se mueven hacia los muros. Los otros dos movimientos son válidos ya que no se mueven hacia el muro y luego ninguno de ellos se encuentra en la listaInterior.

Esos dos nodos válidos se añaden a la listaFrontera, añadiendo primero el nodo hacia abajo y después el nodo hacia la derecha dado que ese es el orden en el que se han creado.

En la siguiente iteración del while, se escogerá el nodo con menor f, dado que la listaFrontera almacenó primero al nodo correspondiente al movimiento hacia abajo y los nodos tienen el mismo coste. Se tomará como nodoAux el nodo correspondiente a moverse abajo.

Este nodo se expandirá de nuevo con cuatro hijos, y se descartán los nodos correspondientes hacia abajo y la izquierda dado que se mueve hacia un muro. Y el movimiento hacia arriba, dado que este movimiento causaría un bucle. Por tanto el único movimiento disponible se almacena en la listaFrontera.

En la siguiente iteración de nuevo se toma el nodo almacenado en la listaFrontera con la menor f, este nodo se corresponde al nodo alamcenado anteriormente dado que es la posición más cercana a pacman.

Varias iteraciones siguiendo los mismos pasos, hasta que el nodo alcanza la posición de pacman. Es ahora cuando se cumple la condición de meta.

Para hallar el movimiento que se realizó en un primer lugar se retroce de nodo en nodo asignando el padre al nodo actual hasta llegar a un nodo cuyo padre sea el nodo inicial. O lo que es lo mismo un nodo cuyo padre no tenga padre. Dado que nuestra clase nodo almacena el movimiento que le llevo a esta posición, se retorna ese movimiento finalizando el algoritmo.

-para pacman

Para evitar reiterarnos, no describiremos una traza de pacman ya que se ha detallado con suficiente claridad esta implementación en el apartado anterior.

Heurística

La heurística se conoce como el conjunto de técnicas y métodos para resolver un problema. En este caso implementar una heurística nos permite descartar aquellos nodos que expandidos no llevan a la solución más óptima. Para ello la heurística nos permite aproximar una solución optimista, es decir, estima una posible solución y siempre lo hace de manera optimista.

Heurísticas utilizadas

Para este problema se pueden utilizar distintas maneras para calcular un valor heurístico.

Para calcular la heurística, la clase nodo cuenta con el método calcH(), a continuación se muestra el código correspondiente a ese método:

```
//método para calcular la heurística y asignarla
void calcH(int n, int m){
```

Vemos que es un método void, pues no retorna nada, si no que queda asignado a la variable h del nodo.

Distancia euclidea

La distancia euclidea nos permite dadas dos coordenadas calcular la distancia entre esas dos coordenadas.

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

La implementación de esta heurística es la siguiente:

```
//distancia euclídea
this.h = (int) Math.sqrt((n - this.n)^2 +
                (m - this.m)^2);
```

Distancia de Manhattan

La distancia de Manhattan o distancia taxi, permite calcular la distancia entre dos puntos. En este caso no es la distancia diagonal si no una distancia en ”L”.

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$

La implementación de esta heurística es la siguiente:

```
//distancia de manhattan
int res1, res2;
res1 = this.n - n;
res2 = this.m - m;
if(res1 < 0){
    res1 = res1 * (-1);
}
if(res2 < 0){
    res2 = res2*(-1);
}
this.h = res1 + res2;
```

Comparación de los diferentes resultados de las distintas heurísticas

Para comparar las diferentes heurísticas nos basaremos en el número de nodos explorados, entenderemos que una heurística será más óptima cuando explore un menor número de nodos, ya que eso quiere decir que el algoritmo encuentra antes el camino de menor coste.

Para poder comparar el número de nodos expandidos, tomaremos como referencia el laberinto5 visto anteriormente y mediremos el número de nodos expandidos por el fantasma antes de realizar el primer movimiento.

Este es el resultado obtenido con la distancia eculídea:

```
ENCONTRADO
NO MODIFICAR ESTE FORMATO DE SALIDA
Coste del camino: 6.0
Nodos expandidos: 26
Camino
. . . . . .
. x . . . . .
. x x . . . .
. . x . . . .
. . x x x . .
. . . . . .
. . . . . .
Camino explorado
-1 -1 -1 -1 -1 -1 -1
-1  0 -1 -1 -1 -1 -1
-1  1  2 -1 -1 -1 -1
-1 -1  3 -1 -1 -1 -1
-1 -1  4  5  6 -1 -1
-1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1
```

Este es el resultado obtenido con la distancia de manhattan:

```
ENCONTRADO
NO MODIFICAR ESTE FORMATO DE SALIDA
Coste del camino: 6.0
Nodos expandidos: 12
Camino
. . . . . .
. x . . . . .
. x x . . . .
. . x . . . .
. . x x x . .
. . . . . .
. . . . . .
Camino explorado
-1 -1 -1 -1 -1 -1 -1
-1  0 -1 -1 -1 -1 -1
-1  1  2 -1 -1 -1 -1
-1 -1  3 -1 -1 -1 -1
-1 -1  4  5  6 -1 -1
-1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1
```

Vemos como con ambas heurísticas se llega a la misma solución. Sin embargo, la heurística que utiliza la distancia de manhattan expande menos nodos.

Por consecuente la distancia de manhattan es una mejor heurística para este problema. Eso se debe a que en el comportamiento de los personajes no se contempla la posibilidad de moverse en diagonal, por lo que la distancia euclidea da un valor demasiado optimista, frente a la distancia de manhattan que se ajusta más al camino real.

Cabe mencionar que con la distancia de manhattan se sigue infraestimando el coste del camino ya que no se tienen en cuenta los obstáculos, es por ello que es un buen candidato para ser una heurística, y en este caso el mejor.

Laberintos de prueba

Además de un laberinto de prueba llamado laberinto4 y laberinto 5.
Se han creado diferentes mapas para comprobar las diferentes casuísticas.

Casuísticas

Pacman bloqueado

Tal y como esta planteado el algoritmo, pacman no puede quedarse quieto, no existe un movimiento que sea el de no moverse. Por consecuente si pacman estuviese rodeado por cuatro paredes pacman, se movería hacia la pared.

Fantasma bloqueado

Lo mismo sucede con el fantasma, si este se encuentra rodeado por paredes, como no puede no moverse, se moverá a una posición no válida.Pacman acorralado

Si pacman está rodeado por tres paredes menos en una posición en la que este un fantasma. Se moverá hacia el fantasma porque es la única posición que no es hacia un muro.

Si pacman está rodea en su cuatro posiciones posibles por cuatro fantasmas, se moverá hacia arriba ya que todas las posiciones tienen el mismo coste hacia los fantasmas, y el primer movimiento es hacia arriba. Y dado que no puede quedarse quieto esa es la única opción.

No hay camino

De no existir un camino entre los fantasmas y pacman.