Practica 3 Búsqueda informada

Los **algoritmos de búsqueda informada** son aquellos que utilizan información adicional para guiar el proceso de búsqueda hacia la solución de manera más eficiente, en contraste con los algoritmos de búsqueda no informada, que exploran el espacio de búsqueda de manera más ciega. En estos algoritmos, la **información heurística** es clave, ya que permite estimar cuán cerca estamos de una solución en cada paso.

Principales algoritmos de búsqueda informada:

1. **Búsqueda voraz (Greedy Best-First Search)**: Este algoritmo siempre elige expandir el nodo que parece más prometedor de acuerdo con una función heurística, generalmente definida como h(n), que estima el costo desde el nodo actual hasta la meta. Sin embargo, al enfocarse únicamente en el valor heurístico, puede ser muy rápido, pero no siempre encuentra la solución óptima, ni garantiza que encuentre una solución si no se gestiona correctamente.
2. **A*** es uno de los algoritmos más populares de búsqueda informada, ya que combina de manera efectiva el costo del camino hasta el nodo actual con una estimación heurística del costo hasta el objetivo. Utiliza una función de evaluación:

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

- Donde:

  - g(n) es el costo exacto desde el inicio hasta el nodo n,
  - h(n) es la estimación heurística del costo desde n hasta la meta.

A* es **completo** y **óptimo** siempre y cuando la heurística h(n) sea **admisible**, es decir, nunca sobreestime el costo real.

- **IDA*** (Iterative Deepening A*): Combina las ventajas de A* y la búsqueda en profundidad iterativa. IDA* ejecuta A* usando límites de costo progresivamente mayores en lugar de construir toda la frontera como A*. Esto lo hace más eficiente en términos de memoria, aunque a costa de algo de tiempo computacional.

- **Búsqueda con haz (Beam Search)**: Es una variación de la búsqueda voraz que limita la cantidad de nodos que se expanden en cada nivel, manteniendo solo un conjunto limitado de los nodos más prometedores. Es útil en aplicaciones donde el rendimiento es más importante que la óptima, como en algunos sistemas de inteligencia artificial y juegos.

Funciones heurísticas:

- **Heurísticas admisibles**: Son aquellas que nunca sobreestiman el costo real para llegar a la meta. Ejemplo: en un problema de laberinto, una heurística admisible podría ser la distancia euclidiana (si no hay obstáculos).

- **Heurísticas consistentes**: Una heurística es consistente si cumple la propiedad de que, para cualquier nodo n y su sucesor n′, el costo estimado desde n a la meta no es mayor que el costo de ir de n a n′ más la estimación desde n′ hasta la meta. Esta propiedad garantiza que A* no necesitará re expandir nodos.

Estos algoritmos de búsqueda informada son fundamentales en diversas áreas, como la inteligencia artificial, la planificación de rutas y los sistemas de recomendación.

Ejemplo de Algoritmo A* en MATLAB

Supongamos que tenemos un grafo representado por un conjunto de nodos y aristas, donde cada arista tiene un costo asociado. Vamos a usar una matriz de adyacencia para representar el grafo y una función heurística simple que será la distancia euclidiana entre los nodos.

Paso 1: Definir el Grafo

Definimos el grafo como una matriz de adyacencia, donde cada valor representa el costo entre nodos. Si no hay conexión directa entre nodos, el valor será infinito.

```
% Definir la matriz de adyacencia (grafo)
nodos = 6; % Número de nodos
A = [0    1    4    inf  inf  inf;
     1    0    2    6    inf  inf;
     4    2    0    3    3    inf;
     inf  6    3    0    2    5;
     inf  inf  3    2    0    2;
     inf  inf  inf  5    2    0];

% Definir los puntos de los nodos (para la heurística, puede ser una
distancia euclidiana)
coordenadas = [0 0;
               1 1;
               2 2;
               3 2;
               4 3;
               5 3];

% Definir el nodo de inicio y objetivo
inicio = 1;
objetivo = 6;
```

Paso 2: Definir la Heurística

Usamos la distancia euclidiana como la heurística h(n).

```
% Función heurística (distancia euclidiana a la meta)
function h = heuristica(nodo, objetivo, coordenadas)
    h = norm(coordenadas(nodo,:) - coordenadas(objetivo,:));
end
```

Paso 3: Implementación del Algoritmo A*

```
% Algoritmo A*
function [camino, costo_total] = A_star(A, inicio, objetivo, coordenadas)
    nodos = size(A, 1);

    % Inicializar costos y listas
    g = inf(1, nodos);      % Costo desde el inicio a cada nodo
    f = inf(1, nodos);      % Costo estimado total (g + h)
    g(inicio) = 0;
    f(inicio) = heuristica(inicio, objetivo, coordenadas);

    % Lista abierta (nodos por explorar) y cerrada (ya explorados)
    abierta = [inicio];
    cerrada = [];

    % Predecesores (para construir el camino final)
    predecesor = zeros(1, nodos);

    while ~isempty(abierta)
        % Encontrar el nodo con menor f en la lista abierta
        [~, idx] = min(f(abierta));
        actual = abierta(idx);

        % Si llegamos al nodo objetivo, construimos el camino
        if actual == objetivo
            camino = [];
            while actual ~= 0
                camino = [actual, camino]; %#ok<AGROW>
                actual = predecesor(actual);
            end
            costo_total = g(objetivo);
            return;
        end

        % Mover el nodo actual de la lista abierta a la cerrada
        abierta(idx) = [];
        cerrada = [cerrada, actual];

        % Examinar los vecinos del nodo actual
        for vecino = 1:nodos
```

```
            if A(actual, vecino) ~= inf && ~ismember(vecino, cerrada)
                tentativo_g = g(actual) + A(actual, vecino);

                if ~ismember(vecino, abierta)
                    abierta = [abierta, vecino];
                end

                if tentativo_g < g(vecino)
                    % Actualizar el costo g y f
                    predecesor(vecino) = actual;
                    g(vecino) = tentativo_g;
                    f(vecino) = g(vecino) + heuristica(vecino, objetivo,
coordenadas);
                end
            end
        end
    end

    % Si no se encuentra camino
    camino = [];
    costo_total = inf;
    disp('No se encontró camino');
end
% Algoritmo A*
function [camino, costo_total] = A_star(A, inicio, objetivo, coordenadas)
    nodos = size(A, 1);

    % Inicializar costos y listas
    g = inf(1, nodos);      % Costo desde el inicio a cada nodo
    f = inf(1, nodos);      % Costo estimado total (g + h)
    g(inicio) = 0;
    f(inicio) = heuristica(inicio, objetivo, coordenadas);

    % Lista abierta (nodos por explorar) y cerrada (ya explorados)
    abierta = [inicio];
    cerrada = [];

    % Predecesores (para construir el camino final)
    predecesor = zeros(1, nodos);

    while ~isempty(abierta)
        % Encontrar el nodo con menor f en la lista abierta
        [~, idx] = min(f(abierta));
        actual = abierta(idx);

        % Si llegamos al nodo objetivo, construimos el camino
        if actual == objetivo
            camino = [];
```

```
            while actual ~= 0
                camino = [actual, camino]; %#ok<AGROW>
                actual = predecesor(actual);
            end
            costo_total = g(objetivo);
            return;
        end

        % Mover el nodo actual de la lista abierta a la cerrada
        abierta(idx) = [];
        cerrada = [cerrada, actual];

        % Examinar los vecinos del nodo actual
        for vecino = 1:nodos
            if A(actual, vecino) ~= inf && ~ismember(vecino, cerrada)
                tentativo_g = g(actual) + A(actual, vecino);

                if ~ismember(vecino, abierta)
                    abierta = [abierta, vecino];
                end

                if tentativo_g < g(vecino)
                    % Actualizar el costo g y f
                    predecesor(vecino) = actual;
                    g(vecino) = tentativo_g;
                    f(vecino) = g(vecino) + heuristica(vecino, objetivo,
coordenadas);
                end
            end
        end
    end

    % Si no se encuentra camino
    camino = [];
    costo_total = inf;
    disp('No se encontró camino');
end
```

Paso 4: Ejecutar el Algoritmo

```
[camino, costo_total] = A_star(A, inicio, objetivo, coordenadas);


disp('Camino encontrado:');
disp(camino);
```

```
disp('Costo total:');
disp(costo_total);
```

Explicación:

- La matriz de adyacencia **A** representa los costos entre nodos.
- La función heurística se basa en la **distancia euclidiana** entre los nodos.
- La función **A_star** implementa el algoritmo A*, manteniendo listas abiertas y cerradas para explorar los nodos de manera informada.
- El algoritmo retorna el **camino óptimo** y el **costo total** desde el nodo inicial al objetivo.

Este ejemplo puede expandirse para representar problemas más complejos y mejorar la eficiencia.

Problema práctico:

Supongamos que tenemos un mapa representado por una cuadrícula de 10x10. Algunas de las celdas de la cuadrícula representan obstáculos, y nuestro objetivo es encontrar el camino más corto desde un punto de inicio hasta un objetivo, evitando esos obstáculos.

Implementación en MATLAB:

1. Definimos el mapa como una matriz donde los valores 1 representan obstáculos y 0 representa un espacio libre.
2. Utilizaremos el algoritmo A* para encontrar el camino más corto.

Paso 1: Definir el mapa y los parámetros del problema

```
% Definir el mapa (10x10) - 0 es libre, 1 es un obstáculo
mapa = [0 0 0 0 0 0 1 0 0 0;
        0 1 1 1 1 0 1 0 1 0;
        0 0 0 0 1 0 1 0 1 0;
        0 1 0 1 1 0 1 0 1 0;
        0 1 0 1 0 0 0 0 1 0;
        0 1 0 1 1 1 1 0 0 0;
        0 0 0 0 0 0 1 1 1 0;
        1 1 1 1 1 0 0 0 0 0;
        0 0 0 0 1 0 1 1 1 0;
        0 0 0 0 0 0 0 0 1 0];
```

```
inicio = [1, 1];     % Punto de inicio (fila, columna)
objetivo = [10, 10]; % Punto objetivo (fila, columna)
```

Paso 2: Definir la heurística (distancia Manhattan)

La distancia de Manhattan es una heurística adecuada para este tipo de problema, ya que en una cuadrícula solo se puede mover en líneas rectas o en ángulo recto.

```
% Función heurística: Distancia Manhattan
function h = heuristica_manhattan(nodo, objetivo)
    h = abs(nodo(1) - objetivo(1)) + abs(nodo(2) - objetivo(2));
end
```

Paso 3: Implementación del Algoritmo A*

```
function [camino, costo_total] = A_star_grid(mapa, inicio, objetivo)
    % Dimensiones del mapa
    [nfilas, ncols] = size(mapa);

    % Crear las matrices de costos
    g = inf(nfilas, ncols);  % Costo g (distancia desde el inicio)
    f = inf(nfilas, ncols);  % Costo f (g + h)

    % Inicializar el costo del nodo de inicio
    g(inicio(1), inicio(2)) = 0;
    f(inicio(1), inicio(2)) = heuristica_manhattan(inicio, objetivo);

    % Inicializar listas abiertas y cerradas
    abierta = inicio; % Lista de nodos por explorar
    cerrada = [];     % Lista de nodos ya explorados

    % Predecesores para reconstruir el camino
    predecesor = zeros(nfilas, ncols, 2);

    % Movimientos posibles (arriba, abajo, izquierda, derecha)
    movimientos = [-1, 0; 1, 0; 0, -1; 0, 1];

    while ~isempty(abierta)
        % Encontrar el nodo con el menor valor f
        [~, idx] = min(f(sub2ind(size(f), abierta(:,1), abierta(:,2))));
        actual = abierta(idx, :);

        % Si hemos llegado al objetivo
        if isequal(actual, objetivo)
```

```
            % Reconstruir el camino
            camino = actual;
            while any(predecesor(actual(1), actual(2), :) ~= 0)
                actual = squeeze(predecesor(actual(1), actual(2), :))';
                camino = [actual; camino]; %#ok<AGROW>
            end
            costo_total = g(objetivo(1), objetivo(2));
            return;
        end

        % Eliminar el nodo actual de la lista abierta y añadirlo a la
cerrada
        abierta(idx, :) = [];
        cerrada = [cerrada; actual];

        % Explorar los vecinos (arriba, abajo, izquierda, derecha)
        for i = 1:4
            vecino = actual + movimientos(i, :);

            % Verificar que el vecino esté dentro del mapa y no sea un
obstáculo
            if vecino(1) > 0 && vecino(1) <= nfilas && vecino(2) > 0 &&
vecino(2) <= ncols
                if mapa(vecino(1), vecino(2)) == 0 && ~ismember(vecino,
cerrada, 'rows')
                    costo_tentativo_g = g(actual(1), actual(2)) + 1; %
Costo de movimiento entre celdas es 1

                    if ~ismember(vecino, abierta, 'rows')
                        abierta = [abierta; vecino]; % Añadir a la lista
abierta
                    end

                    if costo_tentativo_g < g(vecino(1), vecino(2))
                        % Actualizar los costos g y f
                        predecesor(vecino(1), vecino(2), :) = actual;
                        g(vecino(1), vecino(2)) = costo_tentativo_g;
                        f(vecino(1), vecino(2)) = g(vecino(1), vecino(2))
+ heuristica_manhattan(vecino, objetivo);
                    end
                end
            end
        end
    end

    % Si no se encuentra camino
    camino = [];
    costo_total = inf;
```

```
    disp('No se encontró un camino.');
end
```

Paso 4: Ejecutar el Algoritmo

```
[camino, costo_total] = A_star_grid(mapa, inicio, objetivo);


disp('Camino encontrado:');
disp(camino);


disp('Costo total:');
disp(costo_total);


% Visualizar el camino en el mapa
figure;
imagesc(mapa);
hold on;
plot(inicio(2), inicio(1), 'go', 'MarkerSize', 10, 'MarkerFaceColor',
'g'); % Inicio
plot(objetivo(2), objetivo(1), 'ro', 'MarkerSize', 10, 'MarkerFaceColor',
'r'); % Objetivo
plot(camino(:,2), camino(:,1), 'b-', 'LineWidth', 2); % Camino
title('Camino encontrado por A*');
```

Explicación del código:

1. **Mapa:** El mapa de 10x10 tiene obstáculos (valor 1) y áreas libres (valor 0).
2. **Algoritmo A***: El algoritmo explora el espacio de búsqueda usando una función heurística basada en la distancia de Manhattan, que es adecuada para movimientos en una cuadrícula.
3. **Costos:** Se actualizan los costos de camino desde el inicio hacia los nodos vecinos, y se mantiene un registro de los nodos visitados.
4. **Visualización:** Finalmente, se visualiza el camino encontrado en el mapa.

Resultado:

Este código te mostrará el camino más corto desde el nodo de inicio hasta el objetivo, evitando los obstáculos.