Practica 2 Algoritmos de búsqueda no informados

Los algoritmos de búsqueda no informados son técnicas utilizadas en la inteligencia artificial y la informática para encontrar soluciones a problemas de búsqueda sin información adicional sobre el problema, más allá de la estructura del espacio de búsqueda. Estos algoritmos no utilizan ningún conocimiento específico sobre el estado final o sobre el costo de moverse de un estado a otro. Se basan únicamente en la exploración sistemática del espacio de búsqueda.

Principales algoritmos de búsqueda no informados:

1. Búsqueda en anchura (BFS - Breadth-First Search)

Descripción: La búsqueda en anchura explora el espacio de búsqueda nivel por nivel, comenzando desde el nodo raíz y expandiendo todos sus vecinos antes de proceder a los nodos en niveles más profundos. Es un algoritmo que garantiza encontrar la solución óptima si todas las transiciones tienen el mismo costo.

Funcionamiento:

Coloca el nodo inicial en una cola (FIFO).
Extrae el primer nodo de la cola, lo expande y añade sus vecinos (nodos hijos) al final de la cola.
Repite el proceso hasta que encuentra el nodo objetivo o se hayan explorado todos los nodos.

Ventajas:

Siempre encuentra la solución más corta (si las transiciones tienen el mismo costo).

Desventajas:

Puede ser muy costoso en términos de memoria, ya que debe almacenar todos los nodos de cada nivel antes de continuar con el siguiente.

Complejidad temporal: $O(b^d)$, donde:

b es el factor de ramificación (número promedio de hijos por nodo).
d es la profundidad de la solución más cercana.

Complejidad espacial: $O(b^d)$, ya que se deben mantener en memoria todos los nodos del nivel actual.

2. Búsqueda en profundidad (DFS - Depth-First Search)

Descripción: La búsqueda en profundidad explora un camino completo desde el nodo inicial hasta un nodo hoja antes de retroceder y probar otros caminos. Sigue el camino más profundo posible antes de retroceder.

Funcionamiento:

Se coloca el nodo inicial en una pila (LIFO).
Extrae el nodo de la cima de la pila, lo expande y coloca sus vecinos en la pila.
Repite hasta encontrar el nodo objetivo o hasta que se hayan visitado todos los nodos posibles.

Ventajas:

Requiere menos memoria que BFS.
Puede ser más eficiente en problemas donde las soluciones están ubicadas en lo profundo del árbol.

Desventajas:

No garantiza encontrar la solución más corta.
Si no se implementan mecanismos de corte (como límites de profundidad o detección de ciclos), puede entrar en bucles infinitos o explorar ramas muy profundas sin éxito.

Complejidad temporal: $O(b^d)$ (similar a BFS).

Complejidad espacial: O(b*d), ya que solo necesita almacenar la rama actual del árbol y algunos nodos auxiliares.

3. Búsqueda de profundidad limitada (Depth-Limited Search)
Descripción: Es una variante de la búsqueda en profundidad que establece un límite en la profundidad que se puede explorar. Evita caer en bucles infinitos y es útil cuando se sospecha que la solución está a una cierta profundidad.
4. Búsqueda en profundidad iterativa (Iterative Deepening Search - IDS)

Descripción: Este algoritmo combina lo mejor de BFS y DFS. Se ejecuta una búsqueda en profundidad, pero con un límite de profundidad que se va incrementando progresivamente. De esta forma, se garantiza que se exploren primero las soluciones más cercanas, como en BFS, pero con el uso eficiente de memoria de DFS.

Ventajas:

Encuentra la solución óptima.
Usa poca memoria.

Desventajas:

Puede realizar trabajo redundante, ya que expande nodos varias veces en diferentes iteraciones.

5. Búsqueda de costo uniforme (Uniform-Cost Search - UCS)

Descripción: Este algoritmo expande el nodo con el costo acumulado más bajo desde el nodo inicial. Es similar a BFS, pero aquí las transiciones entre nodos pueden tener diferentes costos.

Ventajas:

Garantiza encontrar la solución óptima cuando los costos son distintos.

Desventajas:

Requiere más memoria que BFS o DFS.

Complejidad: Depende del costo de las transiciones entre nodos.

Estos algoritmos son útiles cuando no se dispone de información específica que permita dirigir la búsqueda, lo que los hace fundamentales en problemas de exploración exhaustiva o sin una heurística clara.

Ejemplo de búsqueda en anchura en MATLAB

En este ejemplo, usaremos un grafo representado por una matriz de adyacencia. La matriz de adyacencia es una matriz cuadrada donde el valor en la posición (i, j) es 1 si hay una arista que conecta el nodo i con el nodo j, y 0 si no la hay.
Paso 1: Definimos el grafo con una matriz de adyacencia

```
% Definimos el grafo como una matriz de adyacencia
% Ejemplo de grafo:
% Nodo 1 está conectado a 2 y 3
% Nodo 2 está conectado a 4
% Nodo 3 está conectado a 4 y 5
% Nodo 4 está conectado a 6
% Nodo 5 está conectado a 6
% Nodo 6 no tiene más conexiones

adjMatrix = [0 1 1 0 0 0;  % Nodo 1
             0 0 0 1 0 0;  % Nodo 2
             0 0 0 1 1 0;  % Nodo 3
             0 0 0 0 0 1;  % Nodo 4
             0 0 0 0 0 1;  % Nodo 5
             0 0 0 0 0 0]; % Nodo 6
```

Paso 2: Implementamos el algoritmo BFS

```
function bfs(adjMatrix, startNode)
    % Número de nodos en el grafo
    numNodes = size(adjMatrix, 1);

    % Inicializar una cola para nodos por visitar
    queue = [startNode];

    % Vector para marcar los nodos visitados
    visited = false(1, numNodes);
```

```
    % Marcar el nodo inicial como visitado
    visited(startNode) = true;

    % Iterar mientras haya nodos en la cola
    while ~isempty(queue)
        % Extraer el primer nodo de la cola
        currentNode = queue(1);
        queue(1) = [];

        % Mostrar el nodo actual
        fprintf('Visitando nodo %d\n', currentNode);

        % Revisar los nodos adyacentes
        for neighbor = 1:numNodes
            if adjMatrix(currentNode, neighbor) == 1 && ~visited(neighbor)
                % Si hay una conexión y el nodo no ha sido visitado, agregarlo a
la cola
                queue = [queue, neighbor];

                % Marcar el vecino como visitado
                visited(neighbor) = true;
            end
        end
    end
end
```

Paso 3: Ejecutar la búsqueda en anchura

Ahora ejecutamos la función bfs partiendo desde el nodo 1.

```
startNode = 1;
bfs(adjMatrix, startNode);
```

Explicación del código:
Grafo: El grafo está representado mediante una matriz de adyacencia, donde las filas y columnas representan los nodos, y un valor de 1 en una celda indica que existe una arista entre los nodos correspondientes.
Cola: Usamos una cola (FIFO) para almacenar los nodos que se deben explorar.
Visitados: Marcamos los nodos que ya hemos visitado para no volver a explorarlos.
Recorrido: Extraemos nodos de la cola y añadimos sus vecinos no visitados a la cola hasta que no queden más nodos por explorar.
Salida esperada:

```
Visitando nodo 1
Visitando nodo 2
Visitando nodo 3
Visitando nodo 4
Visitando nodo 5
Visitando nodo 6
```

En este ejemplo, comenzamos desde el nodo 1 y recorremos el grafo en anchura, explorando todos los nodos nivel por nivel.

Ejemplo de búsqueda en profundidad con Matlab
Paso 1: Definir el grafo con una matriz de adyacencia

Este es el mismo grafo que usamos en el ejemplo anterior:

```
% Definimos el grafo como una matriz de adyacencia
% Ejemplo de grafo:
% Nodo 1 está conectado a 2 y 3
% Nodo 2 está conectado a 4
% Nodo 3 está conectado a 4 y 5
% Nodo 4 está conectado a 6
% Nodo 5 está conectado a 6
% Nodo 6 no tiene más conexiones

adjMatrix = [0 1 1 0 0 0;  % Nodo 1
             0 0 0 1 0 0;  % Nodo 2
             0 0 0 1 1 0;  % Nodo 3
             0 0 0 0 0 1;  % Nodo 4
             0 0 0 0 0 1;  % Nodo 5
             0 0 0 0 0 0]; % Nodo 6
```

Paso 2: Implementar el algoritmo DFS recursivo

A diferencia del algoritmo BFS, donde usamos una cola, en la búsqueda en profundidad usamos una pila (LIFO). Esto se puede implementar de manera recursiva en MATLAB.

```
function dfs(adjMatrix, currentNode, visited)
    % Número de nodos en el grafo
    numNodes = size(adjMatrix, 1);

    % Marcar el nodo actual como visitado
    visited(currentNode) = true;

    % Mostrar el nodo actual
    fprintf('Visitando nodo %d\n', currentNode);

    % Recorrer los vecinos no visitados del nodo actual
    for neighbor = 1:numNodes
        if adjMatrix(currentNode, neighbor) == 1 && ~visited(neighbor)
            % Si el vecino no ha sido visitado, hacer la llamada recursiva
            dfs(adjMatrix, neighbor, visited);
        end
    end
end
```

Paso 3: Ejecutar la búsqueda en profundidad

Para iniciar la búsqueda en profundidad, se necesita llamar a la función dfs con el nodo inicial y un vector que lleve el registro de los nodos visitados.

```
% Definir el nodo de inicio
startNode = 1;

% Inicializar el vector de nodos visitados
visited = false(1, size(adjMatrix, 1));

% Llamar a la función de búsqueda en profundidad
dfs(adjMatrix, startNode, visited);
```

Explicación del código:
Grafo: El grafo está representado mediante una matriz de adyacencia, donde las filas y columnas representan los nodos, y un valor de 1 en una celda indica que existe una arista entre los nodos correspondientes.
Visitados: Usamos un vector visited para registrar qué nodos ya hemos explorado, evitando revisitar nodos en bucles.
Recursividad: La búsqueda en profundidad sigue explorando un camino hasta llegar a un nodo sin vecinos no visitados. Luego, retrocede y explora otros caminos. Esto se implementa mediante llamadas recursivas.
Pila implícita: En este caso, MATLAB usa una pila implícita en el manejo de la recursividad. A medida que la función se llama recursivamente, las llamadas se apilan, y el backtracking ocurre cuando ya no hay más nodos vecinos por visitar.
Salida esperada:

```
Visitando nodo 1
Visitando nodo 2
Visitando nodo 4
Visitando nodo 6
Visitando nodo 3
Visitando nodo 5
```

Este es el orden en que se visitan los nodos con la búsqueda en profundidad, comenzando desde el nodo 1. Como se puede ver, explora profundamente un camino ($1 \rightarrow 2 \rightarrow 4 \rightarrow 6$), luego retrocede para continuar con otros nodos no explorados ($3 \rightarrow 5$).