BÚSQUEDA DE ALGORITMOS

Memoria Práctica 2

Asignatura Inteligencia Artificial

Curso 2024/2025

Realizado por:
Lucía Zamudio Cecilia

Índice

1. **Introducción**

Los algoritmos de búsqueda son fundamentales en la resolución de problemas. Podemos clasificarlos para la resolución de la practica en 2 tipos, la búsqueda no informada la cual no tiene una heurística que dirija la búsqueda, el espacio de estados es pequeño y se puede explorar por completo siguiendo una estrategia fija o hasta encontrar la solución, lo cual detendría la búsqueda, y además obtener la solución más optima. Como ejemplos de este tipo tenemos la búsqueda en anchura y la búsqueda en profundidad.
La búsqueda informada utiliza una heurística específica, que estima cuán cerca está cada estado de la meta, y explora aquellos caminos más prometedores para reducir el tiempo y esfuerzo de búsqueda. Por ejemplo: primero el mejor, A*, etc.

OBJETIVO

Como objetivo hay que implementar diferentes tipos algoritmos de búsqueda y comparar su desempeño en la resolución de laberintos. Además, se evaluará la efectividad de cada algoritmo en distintos entornos.

2. **Descripción de los algoritmos**

Algoritmo en Anchura

Para la búsqueda de anchura no se cambia de nivel hasta que no se haya recorrido todo el nivel actual completo.
Para su implementación encontramos el método **void buscarCamino(Nodo inicial, Laberinto lab, int coste)**, en él se crea una cola, para guardar los nodos que vamos a expandir. Primero se inserta el nodo origen en la lista (la raíz) y se marca como visitado, mientras la cola no este vacía se extrae el primer nodo de la cola, se comprueba que el nodo actual sea la salida, si lo es la búsqueda se ha acabado, si no se exploraran a los hijos de ese nodo, los cuales se marcan como visitado y se agregan a la cola, para observarlos en la próxima iteración, hasta que la cola se haya vaciado. **List<int[]> reconstruirCamino(Nodo ultimo),** recorre el camino desde el último nodo(la meta 'S'), hasta el nodo inicial, guardando las coordenadas de ese camino en orden inicio fin. Además posee **void imprimirResultado(Laberinto laberinto)**, en el cual se muestran todos los resultados necesarios para la solución propuesta.

Algoritmo en Profundidad

Para la búsqueda de profundidad, expande un camino al máximo partiendo de la raíz , cuando no puede expandir más una rama, retrocede al nodo más cercano para continuar por ahí la expansión.
Para su implementación se utiliza el método **void buscarCamino(Nodo inicio, Laberinto lab, int coste)**, en el cual se emplea una pila (en lugar de una cola, como en anchura) para almacenar los nodos pendientes de expansión.
Primero se inserta el nodo origen en la pila y se marca como visitado. Mientras la pila no esté vacía, se extrae el nodo en la cima de la pila y se comprueba si es la salida. Si se ha llegado a la meta, la búsqueda finaliza. Si no, se generan los hijos del nodo actual (sus posibles movimientos válidos), se marcan como visitados, y se apilan para continuar la exploración en la siguiente iteración.
Este proceso continúa hasta encontrar la meta o hasta que la pila esté vacía, lo que indicaría que no existe un camino.
Al igual que en búsqueda encontramos los métodos **List<int[]> reconstruirSolucion(Nodo meta)** y **void imprimirResultado(Laberinto laberinto)** que siguen la misma lógica que en el caso anterior.

Para los algoritmos de búsqueda informada hay propuestas 3 heurísticas, la heurística Manhattan que calcula la distancia en línea recta, la cual solo permite movimientos verticales y horizontales.
La heurística Euclidiana que mide la distancia más corta pero esta vez considerante los movimientos diagonales.
Mi heurística parte de la heurística de Manhattan pero tiene una penalización si se encuentra una pared en alguna de las direcciones, siendo esta de 1, si la dirección es hacia abajo o derecha, y 2 si es hacia arriba o izquierda, debido a que la salida del laberinto por lo general, se encontrará cada vez más abajo y a la derecha.

Al igual que en búsqueda no informada encontramos los métodos **List<int[]> reconstruirSolucion(Nodo meta)** y **void imprimirResultado(Laberinto laberinto)** que siguen la misma lógica en todos los algoritmos.
Además incluye los métodos **int calcularPenalizacionPorParedes(Nodo nodo, Laberinto lab),** donde se añade la Penalización mencionada anteriormente. **Y double calcularHeurística(Nodo nodo, Laberinto lab, int tipoHeuristica),** donde se calcularán los 3 tipos de heurísticas.

Algoritmo Greedy Best-First Search (GBFS)

Es un ejemplo de búsqueda informada cuya estrategia prioriza los nodos según h(n) (estimación heurística del coste restante). GBFS no garantiza la solución óptima, ya que ignora el coste acumulado del camino recorrido (g(n)).
Para su implementación, se utiliza el método Resultado **buscarCamino(Nodo inicio, Laberinto laberinto, int tipoHeuristica, int coste)**, el cual emplea una cola con prioridad para almacenar los nodos pendientes de expansión, organizándolos según el valor de h(n).
Inicialmente, se inserta el nodo de origen en la cola y se marca como visitado. Mientras haya nodos en la cola, se selecciona el que tenga la menor estimación heurística y se verifica si es la salida. Si el nodo actual representa la solución, el algoritmo finaliza y devuelve la solución. En caso contrario, se generan sus sucesores, se les asigna un valor h(n) y se añaden a la cola con prioridad para continuar la exploración en futuras iteraciones.
Este proceso continúa hasta encontrar la meta o hasta que la cola esté vacía, lo que indicaría que no existe un camino válido.

Algoritmo A*

El algoritmo A* es una estrategia de búsqueda informada que encuentra el camino óptimo desde un nodo de inicio hasta una meta, evaluando el costo acumulado y una estimación heurística. La prioridad la marca la función de estimación f(n) = g(n) + h(n), donde g(n) representa el costo real desde el inicio hasta el nodo actual, y h(n) es una estimación del costo restante hasta la meta.
Para su implementación, se utiliza el método **Resultado buscarCamino(Nodo inicio, Laberinto laberinto, int tipoHeuristica, int coste)**, el cual emplea una cola de prioridad para almacenar los nodos pendientes de expansión, organizados por su costo total f(n).
El algoritmo comienza insertando el nodo de inicio en la cola y marcándolo como visitado. Mientras existan nodos en la cola, se extrae aquel con el menor valor de f(n). Si el nodo actual es la meta (salida), el algoritmo reconstruye el camino utilizando la matriz de padres y finaliza.
Si no se ha alcanzado la meta, se generan los sucesores del nodo actual, se calcula su g(n), h(n) y f(n), y se añaden a la cola si aún no han sido visitados. Este proceso se repite hasta encontrar la solución o hasta que la cola esté vacía, lo que indicaría que no existe un camino válido.

Algoritmo IDA*

Es un algoritmo de iteración de búsqueda en profundidad con un límite en la búsqueda, el límite lo da una cota máxima sobre el valor de la función.
La implementación se basa en el método **buscarCamino(Nodo inicio, Laberinto laberinto)**, que comienza estableciendo el límite con el valor de h(n) del nodo inicial. Se realiza una búsqueda en profundidad con el límite actual y si no se encuentra la salida, se ajusta el límite para la siguiente iteración.
En cada iteración:

- Se expande el nodo actual si su f(n) está dentro del límite.
- Si se llega a la meta, se reconstruye el camino a través de los nodos padre.
- Si no hay solución dentro del umbral actual, se incrementa el umbral y se reinicia la búsqueda desde el nodo inicial.

Búsqueda en Profundidad con Límite

El algoritmo realiza una búsqueda en profundidad hasta una cierta profundidad máxima (limite).
Para su implementación, se utiliza el método **profundidadConLimite(Nodo actual, int limite, Laberinto laberinto, int coste)**, el cual verifica si un nodo ha alcanzado el límite antes de seguir explorándolo. Si el nodo actual es la salida del laberinto, la búsqueda termina exitosamente, y si no, se generan sus sucesores, marcándolos como visitados y expandiéndolos hasta alcanzar el límite.

Búsqueda en Profundidad Iterativa

Este algoritmo aplica el anterior de forma iterativa, aumentando el límite de profundidad paso a paso hasta encontrar la solución.
Para su implementación, se usa el método profundidadIterativa(Nodo inicio, Laberinto laberinto, int coste), se empieza con un limite de profundidad bajo, se ejecuta la búsqueda de profundidad con el límite, seguidamente se incrementa el límite, y así sucesivamente.

Búsqueda Bidireccional

Es un algoritmo que busca simultáneamente desde el nodo inicial (inicio) y desde el nodo final (salida). Cuando ambos caminos se encuentran, se considera que se ha encontrado una solución.

3. RESULTADOS EXPERIMENTALES

```
Laberinto Generado:
################
#E  # # #      #
### ###        #
##  # # ###    #
# #            #
# # #          #
# # ## # #      #
## #           #
# ##          S#
################
```

MAZE 1

```
Solución encontrada usando algoritmo BUSQUEDA ANCHURA.
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2, 3] [3, 3] [4, 3] [4,
Nodos expandidos: 72.
Tiempo de ejecución: 1126,8 ns.
Números máximos en estructura de datos COLA: 9.
Profundidad máxima alcanzada: 19.
###############
#E..# # #     #
###.###       #
## .# # ###   #
# #...        #
# # #..       #
# # ##.# #    #
## #  .       #
# ##  .......S#
###############
```

Anchura vs Profundidad

```
SOLUCIÓN ENCONTRADA:
Solución encontrada usando algoritmo BUSQUEDA EN PROFUNDIDAD.
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2, 3] [3, 3] [4,
Nodos expandidos: 41.
Tiempo de ejecución: 120,9 ns.
Números máximos en estructura de datos PILA: 26.
Profundidad máxima alcanzada: 25.
################
#E..# # #     #
###.###       #
## .# # ###   #
# #..........#
# # #        .#
# # ## # #....#
## #      .   #
# ##      ...S#
################
```

```
SOLUCIÓN ENCONTRADA:
Solución encontrada usando algoritmo GREEDY BEST-FIRST SEARCH (heurística: MANHATTAN).
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2, 3] [3, 3] [4, 3] [4, 4] [4, 5] [5, 5] [5,
Nodos expandidos: 41.
Tiempo de ejecución: 183,7 ns.
Números máximos en estructura de datos COLA CON PRIORIDAD: 12.
Profundidad máxima alcanzada: 19.
###############
#E..# # #     #
###.###       #
## .# # ###   #
# #...        #
# # #..       #
# # ##.# #    #
## #  .       #
# ##  .......S#
###############
```

MAZE 1
GBFS
MANHATTAN

```
SOLUCIÓN ENCONTRADA:
Solución encontrada usando algoritmo A* (heurística: MANHATTAN).
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2, 3] [3, 3] [4, 3] [4,
Nodos expandidos: 65.
Tiempo de ejecución: 268,1 ns.
Números máximos en estructura de datos COLA CON PRIORIDAD: 10.
Profundidad máxima alcanzada: 19.
###############
#E..# # #     #
###.###       #
## .# # ###   #
# #...        #
# # #..       #
# # ##.# #    #
## #  ..      #
# ##   ......S#
###############
```

MAZE 1
A*
MANHATTAN

```
Laberinto Generado:
##########
#E    # ##
# #  #  ##
# #   ## #
#     ## #
#   ##   #
#  #    ##
#    #   #
###  ###S#
##########
```

MAZE 3
Mi heurística

```
SOLUCIÓN ENCONTRADA:
Solución encontrada usando algoritmo A* (heurística: Heurística propia).
Camino recorrido (x,y): [1, 1] [2, 1] [3, 1] [4, 1] [5, 1] [5, 2] [6, 2] [7,
Nodos expandidos: 37.
Tiempo de ejecución: 214,1 ns.
Números máximos en estructura de datos COLA CON PRIORIDAD: 8.
Profundidad máxima alcanzada: 16.
##########
#E    # ##
#.#  #  ##
#.#   ## #
#.    ## #
#.. ##   #
# .#....##
# ...# ..#
###  ###S#
##########
```

```
SOLUCIÓN ENCONTRADA:
Solución encontrada usando algoritmo GREEDY BEST-FIRST SEARCH (heurística: Heurística propia).
Camino recorrido (x,y): [1, 1] [2, 1] [3, 1] [4, 1] [5, 1] [5, 2] [6, 2] [7, 2] [7, 3] [7, 4] [6,
Nodos expandidos: 19.
Tiempo de ejecución: 77,4 ns.
Números máximos en estructura de datos COLA CON PRIORIDAD: 9.
Profundidad máxima alcanzada: 16.
##########
#E    # ##
#.#  #  ##
#.#   ## #
#.    ## #
#.. ##   #
# .#....##
# ...# ..#
###  ###S#
##########
```

```
Solución encontrada usando IDA*.
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2,
Nodos expandidos: 77362711.
Tiempo de ejecución: 1054432,300 µs.

 Laberinto con camino solución:
################
#E..# # #     #
###.###       #
## .# # ###   #
# #...         #
# # #..        #
# # ##.# #     #
## #  .        #
# ##  .......S#
################
```

```
Ejecutando profundidad iterativa...
Ejecutando profundidad con limite
Solución encontrada.
Camino recorrido (x,y): [1, 1] [1, 2] [1,
Nodos expandidos: 690.
Tiempo de ejecución: 1464,900 µs.
Profundidad máxima alcanzada: 23.
################
#E..# # #     #
###.###       #
## .# # ###   #
# #.......    #
# # #......   #
# # ## # #.   #
## #      .   #
# ##      ...S#
################
```

```
Ingrese el límite de profundidad: 30
Ejecutando profundidad con limite
Solución encontrada.
Camino recorrido (x,y): [1, 1] [1, 2] [1, 3] [2,
Nodos expandidos: 37.
Tiempo de ejecución: 113,400 µs.
Profundidad máxima alcanzada: 29.
################
#E..# # #.... #
###.###..... #
## .# #.### . #
# #.....    . #
# # #..     . #
# # ## # #  . #
## #        . #
# ##        .S#
################
```

```
Ejecutando profundidad bidireccional...
Solución encontrada.
Camino recorrido (x,y): [1, 1] [1, 2] [1,
Nodos expandidos: 31.
Tiempo de ejecución: 159,700 µs.
Profundidad máxima alcanzada: 9.
################
#E..# # #     #
###.###       #
## .# # ###   #
# #.....      #
# # #          #
# # ## # #     #
## #           #
# ##          S#
################
```

4. **PREGUNTAS**

1. ¿Qué algoritmo expande menos nodos?

Entre los algoritmos que encontramos, el que expande menos nodos teóricamente es Búsqueda Bidireccional porque explora desde el inicio y el final simultáneamente, por lo que se reduce el espacio más o menos a la mitad.
En las imágenes podemos comprobar que efectivamente, la búsqueda bidireccional es la que menos nodos ha expandido con un total de 31.

2. ¿Qué algoritmos encuentran la solución óptima? ¿Por qué?

El algoritmo que tiene el camino más corto es A* ya que evalúa tanto el costo acumulado como la heurística estimada, por lo que evita la exploración innecesaria, garantizando la mejor solución. A su vez BFS(búsqueda en anchura), si los costos son iguales, garantiza el camino más corto pero expande muchos nodos, y la búsqueda bidireccional, dónde el inicio y la salida, están bien determinados, es optima**.**

3. ¿Qué diferencias observas teórica y experimentalmente entre los algoritmos de búsqueda en Anchura y Profundidad?

Teóricamente la estructura que usa la anchura es una cola FIFO, mientras que la profundidad utiliza una pila, además la profundidad no tiene porqué encontrar la solución optima, sin embargo la búsqueda en anchura para pequeños laberintos puede proporcionar una solución bastante óptima.
Además la anchura recorre el laberinto nivel a nivel mientras que en profundidad vas recorriendo un camino en profundidad.
Por lo que podemos observar en las imágenes para un mismo laberinto la búsqueda en anchura expande 72 nodos mientras que profundidad 41, la búsqueda en anchura por lo general expandirá más nodos que en profundidad.

4. ¿En el caso de tener un laberinto de grandes dimensiones con el objetivo de encontrar una solución, cual elegirías, Anchura o Profundidad? Justifica tus respuestas.

En el caso de ser un gran laberinto, la búsqueda en profundidad puede ser una mejor elección debido a que usa menos memoria que la búsqueda en anchura, además lo resuelve de manera más rápida, solo que no te garantiza la solución más optima.

5. Estudiar la admisibilidad y optimidad de la heurística propuesta por el alumno. ¿Garantiza encontrar una solución óptima?

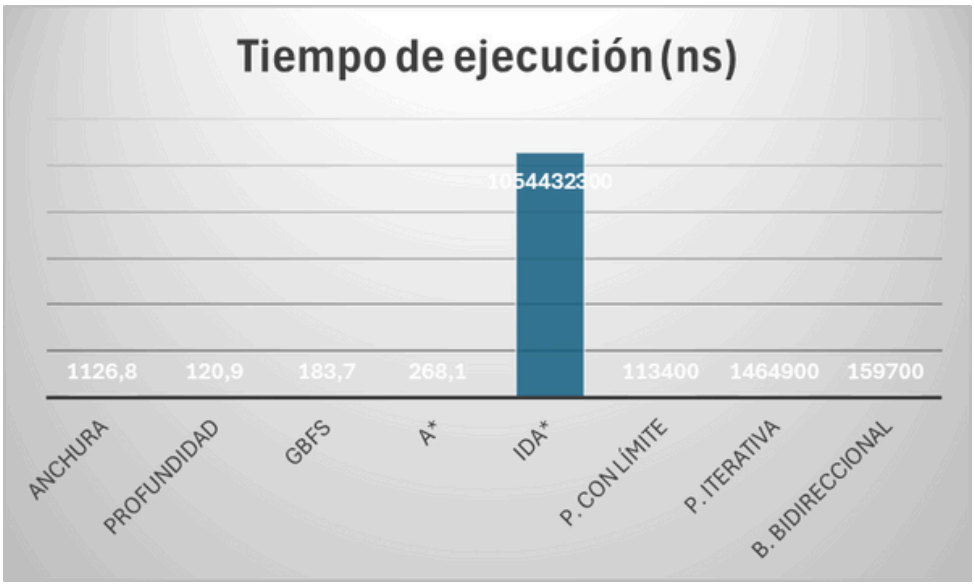
Como tenemos una salida determinada, que siempre se va a encontrar más abajo y a la derecha de la propia entrada, podría ser admisible, ya que favorece estos movimientos, en cuanto a la optimización puede que evite el camino óptimo en algún movimiento y acabe tomando un camino más largo del necesario.

6. ¿Afecta el cambio de coste en los movimientos a la admisibilidad de la heurística Manhattan? Justifica tu respuesta con un ejemplo que se te ocurra.

No afecta si los costos son uniformes, pero si el coste de moverse en vertical es mayor al de moverse en horizontal o al revés, se subestimaría el coste real del camino, por lo que dejaría de ser admisible.

7. Estudiar el tiempo resultante total para la resolución de los laberintos con cada algoritmo desarrollado.

Como podemos observar en el gráfico el algoritmo más rápido es profundidad, y el más lento significativamente es IDA* con heurística Manhattan.



4. **CONCLUSIÓN**

La efectividad de cada algoritmo depende del contexto del problema. A* se destaca por ser el más completo, garantizando la ruta óptima con la menor expansión de nodos cuando se cuenta con una heurística admisible. IDA* es una excelente opción para laberintos con grandes dimensiones o cuando la memoria es limitada, ya que su enfoque de profundización iterativa permite minimizar el consumo de memoria, aunque a costa de realizar más expansiones de nodos.

La búsqueda bidireccional resulta especialmente útil cuando se tiene un buen conocimiento de la salida, ya que al expandir simultáneamente desde el inicio y la meta, reduce significativamente el espacio de búsqueda y acelera el tiempo de ejecución en comparación con otras técnicas.
Por otro lado, BFS garantiza encontrar la ruta más corta en escenarios con costos uniformes, pero es muy costoso en términos de memoria, lo que lo limita en entornos más grandes. DFS y técnicas como la profundidad limitada e iterativa consumen menos memoria y pueden ser útiles cuando el objetivo es explorar soluciones rápidamente, aunque no siempre aseguran encontrar la mejor ruta. Finalmente, GBFS es un algoritmo rápido debido a su enfoque en la heurística, pero su principal limitación es que puede desviarse de la solución óptima, ya que no toma en cuenta los costos acumulados.

En general, no existe un algoritmo único y perfecto, la elección debe basarse en las características del problema en cuestión, como el tamaño del espacio de búsqueda, la disponibilidad de memoria y la importancia de la optimalidad de la solución. Es fundamental evaluar no solo la calidad de la solución, sino también el tiempo de ejecución y los recursos disponibles para tomar una decisión adecuada.