Introducción

En este documento encontrarás un conjunto de problemas y material de soporte sobre el tema de **búsqueda como solución de problemas de planificación**.
La mayoría (no todos) de los problemas aquí indicados requieren de la codificación de un algoritmo de búsqueda en línea con las variantes vistas en la clase teórica. Te será muy útil disponer de tu propio motor de búsqueda y de un framework básico que puedas reutilizar en la resolución de los problemas. Para ello, el ejercicio primero se enfoca en la definición del mismo. En los ejercicios que siguen, utiliza y perfecciona este framework desarrollado aumentando sus capacidades y opciones según las vas necesitando.

Enunciado de los problemas

1) 💻 Construye un modelo base para búsquedas

En este primer ejercicio, el objetivo es iniciar una base de código que luego podemos reutilizar en muchos de los ejercicios siguientes.

Recuerda que desde el punto de vista de código, los elementos base de un algoritmo de búsqueda son:

(a) **Estado** . Es la representación del estado concreto del entorno. Es importante definir una clave, o identificador único del estado que nos permita identificarlo inequivocamente.

(b) **Nodo**. Un nodo es un objeto que da soporte a la estructura de grafo que hemos visto en la teoría que es la base del esquema de búsqueda. Cada nodo contiene como atributos principales una instancia de `Estado` y la información de su ubicación en el árbol de búsqueda (eg. un puntero al nodo padre y otros elementos asociados, en su caso, al proceso de búsqueda, según hemos visto en clase.

La clase **Nodo** es esencialmente independiente del problema de búsqueda que queramos resolver y debe ser, por tanto altamente reutilizable.

(c) **Problema**. La clase problema contiene en la implementación de sus métodos la definición del juego, o búsqueda que queramos resolver, esto es, su *definición*. Esta viene definida por los siguientes métodos que tendrás que re-escribir o sobrecargar:

actions(self,state) Debe retornar la lista de todas las acciones posible, bajo el problema y para el estado dado

isGoal(self,state) Debe retornar True, cuando el estado cumpla las condiciones satisfactorias de la búsqueda, para el problema dado.

transition(self,state,action) Debe retornar el estado resultante de aplicar la acción *action* al estado dado.

cost(self,state,action) Retorna el coste de ejecutar la acción *action* sobre el *state* dado.

Recuerda que el `Problema` ha de ser inicializado con un `Nodo` inicial, o raíz

(d) **SearchAgent** Esta clase anima el algoritmo de búsqueda. Como hemos visto, los diferentes algoritmos (BFS, DFS, A*) esencialmente dependen de la gestión de la cola interna que gestiona la **Frontera** donde se acumulan los nodos que contienen los estados visitados y no expandidos en base a las acciones posibles[1].

En consecuencia, además de otros métodos internos `SearchAgent` requerirá de:

[1] Ver `https://docs.python.org/es/3/library/queue.html` para una discusión sobre las colas en Python.

__init__(self, problem, policy) donde `problem` ha de ser una instancia de `Problema` adecuadamente creada, y donde `policy` indica el tipo de cola con que dotar a la instancia del `SearchAgent`

solve(self) Que aplica el mecanismo de búsqueda sobre el problema asignado, y con la política fijada por el tipo de cola asociada. Este es el método clave del algoritmo de búsqueda. Este método debe devolver `None` en caso de no encontrar nada, o bien el nodo que cumple la condición de éxito fijada en su problema asociado.

Como modelo e inspiración para la implementación del método `solve` consulta la sección 3.3 del libro de Russel& Norvig.

La herencia desde unas clases base que implementen el modelo, caracterizándolo para cada caso es la mejor manera de definir y reutilizar este framework.

2) 💻 Aspiradora de dos habitaciones

Re-implementa el modelo simple de la aspiradora de dos habitaciones utilizando el algoritmo básico de las búsqueda.

- ¿Qué ocurre con el estado final ?

3) 💻 Granjero y cía en el río

Re-implementa con el motor de búsqueda, un modelo de búsqueda de la solución del problema anterior del granjero que cruza el río.

Considera:

- los estados iniciales, finales y las transiciones, son obvios. Puedes reutilzar en el estado la representación que hiciste para el agente Reflex anterior.
- ¿Cómo evitamos volver a situaciones anteriores y entrar en un ciclo indefinido?
- Una vez que has dado con el nodo exitoso, que contiene la solución final, ¿Cómo muestras la solución que lleva a él?[2]
- Prueba los modelos DFS y BFS, observa las diferencias si las hubiera.

4) 💻 Torres de Hanoi

Para convencerte a ti mismo del poder general de la búsqueda volvamos al viejo problema de las Torres de Hanoi que recordarás del año pasado[3]

[2] Quizás quieras extender la clase Nodo con un método que muestra el path hasta la solución inicial.

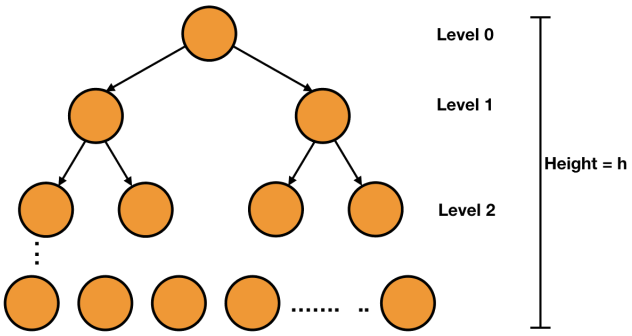
[3] Puedes consultar Wikipedia en https://es.wikipedia.org/wiki/Torres_de_Han%C3%B3i para más detalles sobre el problema.

De nuevo, reutilizando el código base de búsqueda[4]:

- Modeliza el estado, las posible acciones en cada caso, y el estado final, todo ello para una pila de n fichas.
- Busca la solución buscando en el árbol que emana de la posición, y prueba ambos modelos de busqueda.
- Para un caso de n bajo, muestra la solución como la secuencia de acciones que llevan desde las situación inicial hasta la solución final.
- Como métricas del algoritmo, contabiliza (¿cómo?) el número de estados diferentes explorados y la longitud de la secuencia que lleva a la solución, todo ello para cada uno de las "políticas" de búsqueda. ¿Cómo se comparan con la solución óptima que puedes ver en Wikipedia?

5) Algo de Teoría y Complejidad

Utilizando un arbol binario genérico (como el de la figura):



a Estima en el peor escenario posible, el orden de magnitud de las operaciones necesarias para encontrar una solución utilizando, los siguientes modelos de búsqueda:
   - Depth-first search.
   - Breadth-first search.

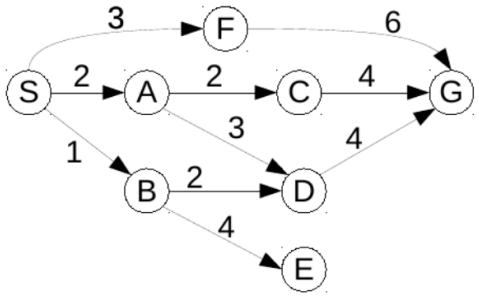
b Repite el problema anterior pero, en lugar del peor escenario posible, teniendo en cuenta que la profundidad típica de la solución menos profunda es d (y la profundidad máxima sigue siendo h).

c ¿Cómo se generaliza este resultado al caso de un árbol con una media de b hijos por nodo?

6) Diferentes Tipos de Búsquedas

El gráfico de la siguiente figura muestra el espacio de estados de un problema de búsqueda hipotético. Los estados se indican con letras y el costo de cada acción se indica en la arista correspondiente. La tabla anexa muestra el valor obtenido de una función heurística (que no conocemos, ni falta que nos hace), considerando G como el estado objetivo.

[4]Daremos mejor puntuación a aquéllos que utilicen la herencia como método de reuso de código



S	A	B	C	D	E	F	G
6	4	5	2	2	8	4	0

a Considerando S como el estado inicial, resuelve el problema de búsqueda anterior usando:
   i. Búsqueda en profundidad (sin tener en cuenta el costo).
   ii. Búsqueda en anchura (sin tener en cuenta el costo).
   iii. Búsqueda de avariciosa con la heurística de la figura.
   iv. Búsqueda A^* con la heurística de la figura.

b Rellena una tabla como la del ejercicio 2) para cada caso.

7) Analizando, Paso a Paso, el algoritmo de búsqueda

Para cada uno de los ejercicios de búsqueda anteriores, rellena unas cuantas filas de la tabla adjunta, ejercitando el proceso de búsqueda de forma manual.

Step	Current Frontier	Removed (s) Node/State	Action(s)	Transition(a,s)	Explored

Table 1: Proceso de búsqueda

8) Otro Clásico: "Slide Puzzle"

Este lo vimos brevemente en clase. Es un problema clásico en el algoritmo de búsqueda. Ver la página 86 del libro de Russel&Norvig para algún detalle más.

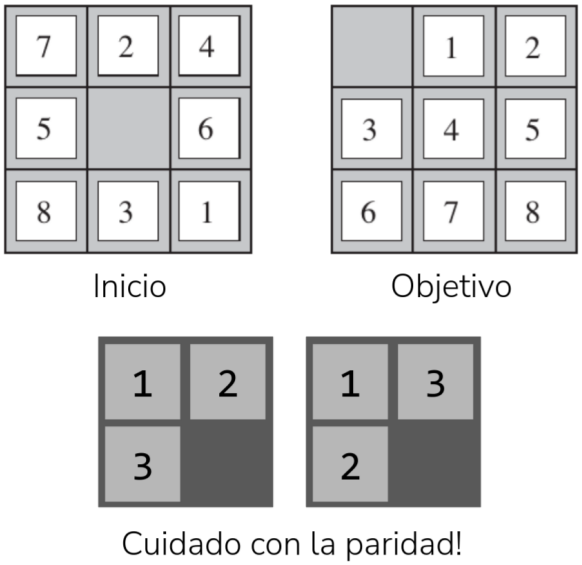


Figure 1: SlidePuzzle

Pasos lógicos:

- Realiza la abstracción del problema y representa el estado de la manera más eficiente posible.
- La acciones posibles son obvias. Se trata de desplazar el espacio libre sobre cualquiera de las celdas adjacentes. (O, más bien, mover cualquiera de las fichas adyacentes al espacio libre sobre éste). El modelo de transición es por tanto muy sencillo.
- Recuerda que no todos los estados iniciales son resolubles (Paridad).
- ¿Qué métricas, o heurísticas puedes imaginar para acelerar el proceso de búsqueda mediante una cola priorizada (A^*)?

9) Navegando el laberinto

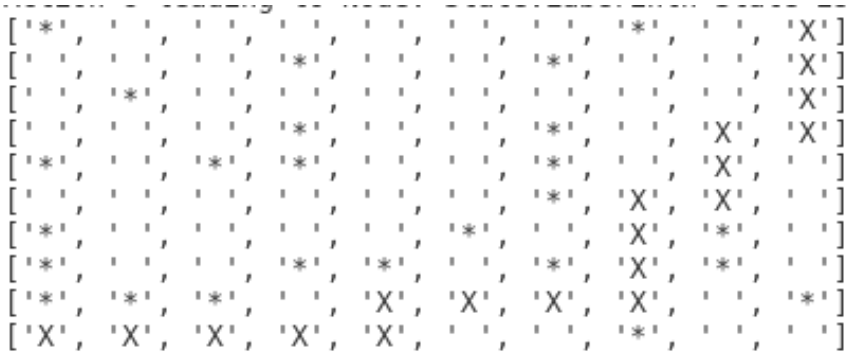


Figure 2: Laberinto

Considera una tablero bidimensional de tamaño $n \times m$ con un conjunto de posiciones $b < (n \times m)$ arbitrariamente elegidas que están prohibidas. Ajusta la selección de las mismas, tal que la posición inicial $(0, 0)$ y la final $(n - 1, m - 1)$ no están bloqueadas.

La visibilidad de tu entorno (límites exteriores y obstáculos) se limita a las celdas adyacentes a tu posición, lo que permite eliminar las posibles acciones (desplazamientos) no válidas. Ver la figura incluida.

Utilizando el algoritmo, y el framework de búsqueda del que ya debes disponer:

- Obtén un camino desde el inicio al final, evitando los obstáculos.
- Utiliza ambos métodos, DFS, y BFS, evitando loops. Compara las soluciones
- Utiliza una función de coste que asegure el camino más corto entre la entrada y la salida
- ¿ Qué heurística puedes utilizar para usar el modelo A* ?
- **BONUS TRACK.** Considera los obstáculo no de forma aleatoria, sino como barreras, u obstáculos sólidos, lo que convierte tu espacio en un laberinto. ¿Funciona tu algoritmo?

10) 💻 Navegando el laberinto en busca de Oro !

Toma el ejercicio anterior, que te ha enseñado a navegar laberintos.

Busquemos ahora una función de optimización más compleja en la que buscamos recoger la máxima cantidad de oro en el camino desde la entrada a la salida.

El oro se encuentra distribuido en una serie de coordenadas i, j desconocidas para el agente y distintas de los obstáculos. Al igual que los obstáculos, tu agente percibe el oro sólo cuando está en la casilla adyacente a su ubicación, y sólo lo puede recoger desplazándose a la ubicación del mismo.

Al recoger el oro, tu agente obtiene una recompensa $g_{i,j}$ positiva, mientras que cada desplazamiento conlleva un coste negativo de -1.

El objetivo es, comenzando en la casilla de partida $(0, 0)$, salir por la casilla $(n-1, m-1)$ con la función de coste acumulada de costes y recompensas máxima.

II) 💻 Metro Navigator: un problema de búsqueda

Utiliza los algoritmos de búsqueda que has visto en clase para determinar el camino óptimo entre dos estaciones cualquiera del metro de Madrid.

La información básica de la red del metro de Madrid, extraída de la web del Consorcio Regional de Transportes de Madrid[5] la encontrarás en el directorio `search` del repositorio en github de la asignatura, que encontrarás en https://github.com/juanclaudioagui/FIA23-24-Alumnos/tree/main/search

Te será útil reconocer que hay varios conceptos que forman parte de la abstracción del problema. Estos son la estación de metro, los andenes (que forman parte de una línea y de una estación, con un sentido dado) y las líneas que unen en secuencia andenes con un sentido y e identificador de línea común.

Aplica una función de costo a cada tramo entre estaciones, así como a la bajada desde la estación a los andenes y a la subida desde éstos a la estación. Utiliza el motor de búsqueda que ya tendrás desarrollado y podrás hallar el camino que te lleve lo más rápido posible a clase (Desde una estación cualquiera a un objetivo como la estación de ARGUELLES, por ejemplo).

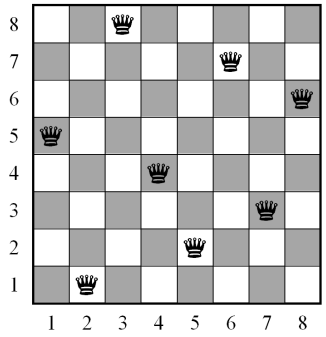
Considera:

- Utiliza el motor de búsqueda para obtener una solución cualquiera, aunque no sea ésta necesariamente la más rápida o corta.
- ¿Cómo te aseguras que es la solución óptima (i.e. la más rápida)?
- Evalúa la bondad de los algoritmos anteriores tomando como referencia el número de andenes explorados en el proceso de búsqueda.
- Piensa en una función heurística que evalúe la distancia al objetivo y aplica el algoritmo A^* para optimizar el proceso de búsqueda. Compara la métrica anterior de eficiencia del algoritmo con los casos anteriores.

[5] https://datos.crtm.es/maps/f3859438e5504a6b9ca745880f72ef1b/about

12) 💻 Búsqueda con Constraints 1: El problema de las 8 reinas

Considera el problema de las ocho reinas[6]



Utiliza el framework de búsqueda para buscar la solución, añadiendo de forma incremental reinas de forma sucesiva al tablero y utilizando los ”constraints” o condicionantes del problema para limitar el número de acciones legales para saltar de un estado con i reinas a un estado con $i + 1$ reinas.

Puede que llegues a situaciones donde sea imposible colocar la siguiente reina, esto es, las condiciones limitan el conjunto de acciones posibles. Entonces tendrás que hacer **backtraking**

- ¿Qué modelo de búsqueda es la apropiada aquí ?
- Generalízalo para n reinas. ‘?Hasta que valor de n puedes escalar en ordenador personal ?
- **Extra !**: Generalízalo a un número arbitrario n.
- Analiza la complejidad de la búsqueda para en función de n

13) 💻 Búsqueda con Constraints 2: Sudoku

Una vez que sepas resolver búsquedas con propagación de constraints (que esencialmente no hacen más que reducir drásticamente la complejidad del árbol de búsqueda) puedes atacar el sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Para ello algunas pistas:

- Considera un modelo eficiente de representación del estado del sudoku (los hay de múltiples tamaños)
- Antes de nada, en cada estado, aplica las reglas del sudoku, para ver si alguna celda no especificada viene ya fijada por las condiciones del mismo tras la fijación de una de las celdas.

[6] Ver https://en.wikipedia.org/wiki/Eight_queens_puzzle.

- Prioriza las acciones a tomar buscando tomar opciones sobre las celdas con el menor número de opciones
- backtrack cuando sea necesario.
- ?Como estimarías, a priori, la complejidad de un sudoku, antes de acometer su resolución ?

Generar un sudoku no es tarea inmediata. Puedes probar la sección de pasatiempos de El País (https://elpais.com/juegos/sudokus/. También los puedes encontrar con diversos grados de dificultad en https://sudoku.com/es