SISTEMAS INTELIGENTES

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Type text here

Práctica 1:

"RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA EN UN ESPACIO DE ESTADOS"

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Valencia, Septiembre 2023

1. Introducción

El objetivo de la práctica consiste en evaluar la eficiencia, coste temporal y espacial de distintas estrategias de búsqueda aplicadas al juego del 8-puzzle. Para ello se proporciona el programa *puzzle*[1] implementado en Python. Para instalar y ejecutar el programa se deben realizar las siguientes acciones:

1. Copia el fichero **puzzle.zip** que está en PoliformaT, en la carpeta rRecursos/Practica 1 en tu cuenta
2. Descomprime el fichero**.** Al descomprimirlo se creará un directorio *puzzle* donde se ubican los ficheros fuente.
3. Desde un entorno de desarrollo de Python (por ejemplo **spyder**, disponible en los laboratorios) o desde un terminal (si tiene Python instalado en el PATH) ejecuta el programa **interface.py**.

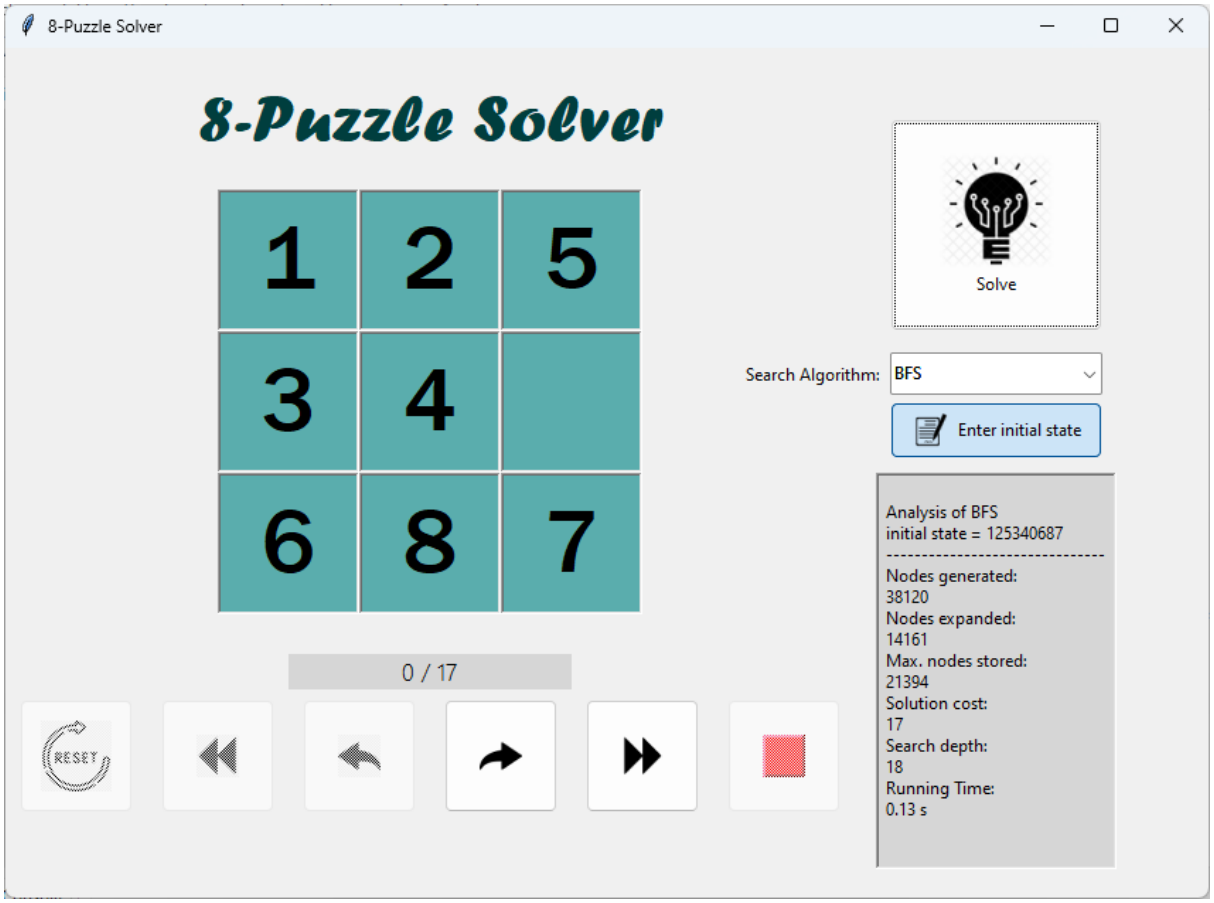


Figura 1 Interfaz principal del programa del puzzle

4. Pulsa en el botón **'Enter initial state'** e inserta el hecho inicial como una secuencia de números donde la casilla blanca se representa como un 0. El estado que aparece en la figura 1 se introduciría como 125340687

[1] Ampliación del programa del mismo nombre desarrollado por Adham Mohamed Alya et al.

5. Selecciona la estrategia de resolución deseada en la pestaña desplegable '**Search Algorithm'**.
6. Si se selecciona una estrategia que requiere un nivel de corte de profundidad se abrirá una ventana donde se podrá introducir dicho valor.
7. Pulsa el botón '**Solve'.**
8. Se mostrarán los datos de la ejecución del proceso de búsqueda realizado, y, además, se puede ver el camino de la solución pulsando sobre los botones debajo de la ventana del puzzle.

Para poder hacer una comparativa entre las distintas estrategias de búsqueda, emplearemos siempre el mismo estado objetivo.

1	2	3
8		4
7	6	5

Nota 1: Si al intentar resolver una configuración del puzzle aparece el mensaje '**Cannot solve'**, eso indica que la configuración introducida no se puede resolver mediante una combinación de movimientos de las piezas del puzzle en las cuatro direcciones permitidas: arriba, abajo, derecha e izquierda.

Nota 2: En las estrategias que requieren la introducción de un nivel máximo de profundidad, se obtendrá el mensaje '**The state you entered is unsolvable'** si el nivel introducido es inferior al nivel de la solución óptima.

2. Aspectos generales del juego del puzzle

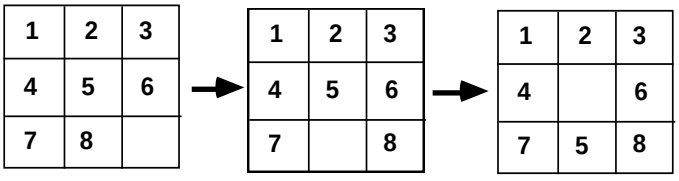
En este apartado se resumen brevemente los aspectos más importantes del juego del puzzle.

2.1. Representación

El juego del puzzle se representa sobre un tablero de 3*3 casillas. 8 de las casillas contienen una pieza o ficha que se puede deslizar a lo largo del tablero horizontal y verticalmente. Las fichas vienen marcadas con los números del 1 al 8. Hay una casilla libre en el tablero que permite los movimientos de las fichas.

El número total de estados o configuraciones del problema que se puede generar en el juego del puzzle es 9! = 362.880 estados.

Ejemplos de movimientos:



El **objetivo** del problema es obtener, como solución, la secuencia de 'movimientos del cuadro vacío' que transforma el estado inicial al estado objetivo:

1	2	3
8		4
7	6	5

2.2. Operadores

Los operadores del juego del puzzle se establecen como los movimientos del cuadro vacío. Estos movimientos son: arriba, abajo, derecha, izquierda. Nótese que, sólo cuando el cuadro vacío está en la casilla central, se podrán realizar los 4 movimientos. Si el cuadro vacío está en una esquina entonces el número de movimientos válidos es 2; en cualquier otro caso existen 3 posibles movimientos para el cuadro vacío.

3. Estrategias de búsqueda.

Las estrategias de búsqueda que ofrece el programa *puzzle* son:

BFS. Esta es la estrategia de ***Anchura*** que utiliza $f(n)=g(n)$ para la expansión de nodos, donde $g(n)$ es el factor coste asociado al nivel de profundidad del nodo n en el árbol de búsqueda.

DFS (Graph Search). Esta estrategia implementa una búsqueda en ***Profundidad*** GRAPH-SEARCH. Para ello, utiliza la función $f(n)=-g(n)$, donde $g(n)$ es el factor coste asociado al nivel de profundidad del nodo n en el árbol de búsqueda.

DFS (Bactracking). Esta estrategia implementa una búsqueda en ***Profundidad*** con backtracking o ***Profundidad*** TREE-SEARCH. Para ello, utiliza la función $f(n)=-g(n)$, donde $g(n)$ es el factor coste asociado al nivel de profundidad del nodo n en el árbol de búsqueda. A diferencia de DFS (Graph Search), esta estrategia solo mantiene en memoria los nodos explorados que pertenecen al camino actual (lista PATH), el resto de nodos explorados no se mantienen en la lista CLOSED sino que se eliminan.

Voraz (Manhattan). Esta es una estrategia que implementa un *Algoritmo voraz* siguiendo la función $f(n)=D(n)$, donde $D(n)$ es la distancia de Manhattan.

ID. Esta estrategia implementa una búsqueda por ***Profundización Iterativa*** (Iterative Deepening). Se trata de realizar sucesivas búsquedas en profundidad con backtracking hasta que se alcanza la solución. Cada nueva búsqueda incrementa el nivel de profundidad de la exploración en 1.

A* Manhattan. Esta es una búsqueda de tipo A que utiliza la distancia de Manhattan como heurística; la expansión de los nodos se realiza siguiendo la función f(n)=g(n)+D(n), donde g(n) es el factor coste correspondiente a la profundidad del nodo n en el árbol de búsqueda y h(n)=D(n) es la distancia de Manhattan, es decir, la suma de las distancias en horizontal y vertical de cada ficha del puzzle desde su posición actual a la posición que ocupará en el objetivo.

A* Euclidean: Esta es una búsqueda de tipo A que utiliza a distancia Euclídea como función heurística; la expansión de los nodos se realiza siguiendo la función f(n)=g(n)+D(n), donde g(n) es el factor coste correspondiente a la profundidad del nodo n en el árbol de búsqueda y h(n)=D(n) es la distancia Euclídea, es decir, la suma de las distancias en línea recta de cada ficha del puzzle desde su posición actual a la posición que ocupará en el objetivo (raíz cuadrada de la suma de los cuadrados de las distancias horizontal y vertical).

IDA* Manhattan. Esta estrategia de búsqueda implementa una búsqueda IDA* donde el límite de la búsqueda viene determinado por el valor-f, es decir, por la función f(n)=g(n)+D(n), donde g(n) es el factor de coste y h(n)=D(n) es la distancia de Manhattan. El límite de una iteración i del algoritmo IDA* es el valor-f más pequeño de cualquier nodo que haya excedido el límite en la iteración anterior i-1.

NOTA IMPORTANTE:

Las estrategias DFS (Graph Search) y DFS (Backtracking) funcionan con un nivel máximo de profundidad. Si el nivel máximo actual es menor que el nivel de la solución óptima entonces la estrategia no encontrará solución.

4. Código a modificar.

Cuando se quiere introducir una nueva estrategia de búsqueda, son varias las zonas del código que debes modificar o añadir.

4.1 Fichero 'interface.py'

- Añadir el nombre de la nueva función de búsqueda para que aparezca en la lista desplegable de estrategias de búsqueda:

Función `def __init__(self, master=None)`: Líneas 88-89 (aprox.):

```
self.algorithmbox.configure(cursor="hand2", state="readonly",
                            values=('BFS',   'DFS   (Graph   Search)',   'DFS
                           (Backtracking)',  'Voraz  (Manhattan)',  'ID',  'A*
                          Manhattan', 'A* Euclidean', 'IDA* Manhattan'))
```

- Si la estrategia requiere una profundidad máxima, hay que añadir el nombre de la función utilizado en el punto anterior en la lista que se utiliza solicitar la máxima profundidad:

Función `selectAlgorithm`, línea 248 (aprox):

```
  if algorithm in ['DFS (Graph Search)', 'DFS (Backtracking)']:
   cutDepth = int(simpledialog.askstring('Cut depth', 'Please, enter your max
              depth'))
```

- Realizar la llamada a la función de búsqueda. Cualquier estrategia de búsqueda ***sin bactracking*** usa la misma llamada `main.graphSearch` pero con diferentes parámetros. La modificación debe realizarse en la función `solveState` (línea 381, aprox) añadiendo un nuevo caso para la nueva estrategia con la llamada a la función graphSearch. Ejemplo de A* Manhattan:

```
elif str(algorithm) == 'A* Manhattan':
     main.graphSearch(initialState,main.function_1,main.getManhattanDistance)
     path, cost, counter, depth, runtime, nodes, max_stored,memory_rep = \
          main.graphf_path, main.graphf_cost, main.graphf_counter,
          main.graphf_depth,          main.time_graphf,          main.node_counter,
main.max_counter,
          main.max_rev_counter
```

La tercera línea es la recolección de resultados y es igual para todas las heurísticas *sin bactracking*. Los parámetros que se le pasan a la función `graphSearch` por orden son:

- Estado inicial del puzzle a resolver. Se almacena en la variable `initialState`.

- Función que calcula el valor $g(n)$. Hay tres funciones predefinidas que se pueden usar:
  - `function_1`. Devuelve 1, representa el coste habitual de un movimiento en el puzzle
  - `function_0`. Devuelve 0, se usa para la estrategia voraz donde no se tiene en cuenta $g(n)$
  - `function_N`. Devuelve -1. Usado en estrategias de profundidad donde los nodos más profundos son más prioritarios

- Función que calcula $h(n)$. Aquí se hace la llamada a la función específica que calcula $h(n)$ según la estrategia, como puede ser `getManhattanDistance`, `getEuclideanDistance` o las nuevas heurísticas que se implementen.

- Profundidad máxima del árbol. Es un parámetro opcional; si se utiliza, el valor se almacena en la variable `cutDepth`. Solo se usa para estrategias que necesiten una profundidad de corte. Si no es así, se omite.

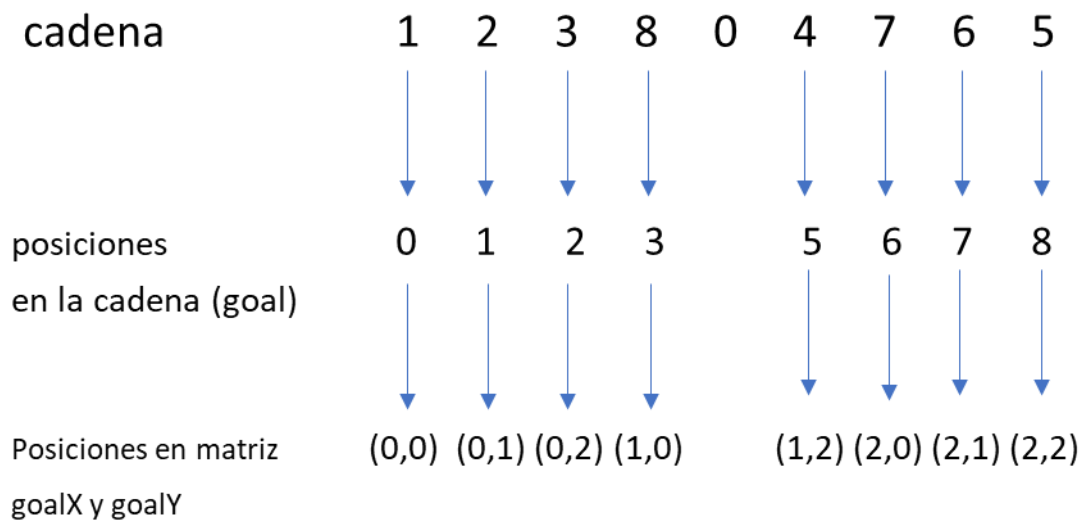
4.2 Fichero 'main.py'

En este fichero hay que incluir una función que implemente la nueva función heurística. A modo de ejemplo explicamos la función `getManhattanDistance` ya incluida en el código:

```
def getManhattanDistance(state):  #state es el estado actual a evaluar (en forma de cadena texto)
  tot = 0
  for i in range(1,9):                #recorre las 8 piezas
    goal = end_state.index(str(i)) #posición de la pieza i en el estado final (end_state)
    goalX = int(goal/ 3)          #Pasamos el end state de cadena a una matriz de 3x3
    goalY = goal % 3              #fila de la pieza (goalX) y columna (goalY)
    idx = state.index(str(i))     #misma operación pero para la fila (idx) y columna (idy)
   itemX = int(idx / 3)           #que ocupa esa pieza en el estado actual
    itemY = idx % 3
   tot += (abs(goalX - itemX) + abs(goalY - itemY)) #cálculo de distancias
 return tot
```

La función recibe en `state` el estado actual (en formato cadena) a evaluar. Para cada una de las 8 piezas (el espacio en blanco se representa con un 0 pero no es una pieza) se calcula la distancia desde la posición actual de esa pieza a su posición en el estado final (representado en la variable `end_state`). Las operaciones intermedias son para para calcular las posiciones de cada pieza en una matriz de 3x3 como en el puzzle a partir de la representación lineal en una cadena.

Por ejemplo, para el estado final (`end_state`), la transformación de representación lineal (cadena) a matriz sería:



TRABAJO A REALIZAR

En este apartado se expone la tarea que tiene que realizar cada grupo de prácticas (bien individual o un grupo de dos personas como máximo) sobre el programa del 8-puzzle.

Se pide implementar dos funciones heurísticas y responder a una serie de preguntas. El día del examen de prácticas todos los estudiantes deben subir a su tarea:

- El código Python de cada una de las dos funciones heurísticas
- Un fichero de texto con las respuestas a las preguntas formuladas

En el caso de un grupo de prácticas de dos personas, ambas subirán los mismos ficheros a sus correspondientes tareas. **NOTA: Indicad claramente los nombres de las dos personas en el fichero de texto que contiene las respuestas a las preguntas formuladas.**

1. Implementa la función heurística **PIEZAS DESCOLOCADAS** que se ha visto en clase de teoría.

2. Implementa la función heurística **SECUENCIAS** que se describe a continuación:

   Secuencias. $f(n)=g(n)+3*S(n)$, donde $g(n)$ es el factor coste asociado al nivel de profundidad del nodo n en el árbol de búsqueda y $h(n)=3*S(n)$.

   $S(n)$ es la secuencia obtenida recorriendo sucesivamente las casillas del puzzle, excepto la casilla central, y anotando 2 puntos en la cuenta por cada ficha no seguida por su ficha sucesora y 0 puntos para las otras; si hay una ficha en el centro, se anota 1. Ejemplo:

2	8	3
1	6	4
7		5

Para la ficha 2:	Sumar 2 (porque no va seguida de su ficha sucesora 3)
Para la ficha 8:	Sumar 2 (porque no va seguida de su ficha sucesora 1)
Para la ficha 3:	Sumar 0 (porque sí va seguida de su ficha sucesora 4)
Para la ficha 4:	Sumar 0 (porque sí va seguida de su ficha sucesora 5)
Para la ficha 5:	Sumar 2 (porque no va seguida de su ficha sucesora 6)
Para la ficha 0:	Sumar 2 (se sumará 2 puntos siempre que no esté en la casilla central)
Para la ficha 7:	Sumar 2 (porque no va seguida de su ficha sucesora 8)
Para la ficha 1:	Sumar 0 (porque va seguida por su ficha sucesora 2)
Para la ficha 6:	Sumar 1 (porque es la ficha central).
TOTAL	**11 puntos**

3. Responde a las siguientes preguntas. Para ello se recomienda ejecutar varias configuraciones del puzzle y anotar los resultados de todas las estrategias de búsqueda que se muestran en el programa, además de las dos nuevas funciones heurísticas implementadas (Descolocadas y Secuencias). A modo de ejemplo, se puede ejecutar las siguientes configuraciones propuestas (se recomienda ejecutar más configuraciones para analizar detalladamente el comportamiento de las estrategias):

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

	BFS	**DFS (GS)**	**DFS (Back)**	**Voraz**	**ID**	**A* Manh**	**A* Euclid**	**IDA* Manh**	**Desc**	**Secuen**
Nodes generated	8761	3230	3539	494	20086	215	286	307	503	114
Nodes expanded	3250	1812	2269	185	12726	79	106	196	186	39
Max nodes stored	5118	1817	27	298	27	135	174	22	300	76
Solution cost	14	14	14	34	14	14	14	14	14	14
Search depth	15	14	14	34	14	14	14	14	14	14
Running time	0.06	0.03	0.02	0.02	0.08	0.01	0.02	0.02	0.02	0.00

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

	BFS	**DFS (GS)**	**DFS (Back)**	**Voraz**	**ID**	**A* Manh**	**A* Euclid**	**IDA* Manh**	**Desc**	**Secuen**
Nodes generated	233687	158233	435995	82	2198820	706	2781	1049	17464	1543
Nodes expanded	86871	60265	279820	30	1390965	266	1028	882	6452	465
Max nodes stored	110728	60275	43	53	43	428	1650	35	10067	902
Solution cost	22	22	22	22	22	22	22	22	22	26
Search depth	23	22	22	22	22	22	22	22	22	29
Running time	1.61	1.06	1.72	0.00	8.44	0.02	0.10	0.03	0.31	0.04

3.1. La estrategia de búsqueda implementada con la función heurística Secuencias, ¿es un algoritmo A*? Justifica la respuesta.
3.2. Compara la estrategia A* Manhattan con Secuencias e indica cuál de las dos estrategias devuelve mejores soluciones (calidad de la solución y coste de la búsqueda).
3.3. Compara la estrategia Descolocadas con Secuencias e indica cuál de las dos estrategias devuelve mejores soluciones (calidad de la solución y coste de la búsqueda).

3.1 Sí, puesto que utiliza una función de evaluación de la forma f(n) = g(n) + h(n), donde g(n) es el factor coste asociado al nivel de profundidad del nodo n en el árbol de búsqueda y h(n)=3*S(n).

3.2 En el primer ejemplo dado, ambas estrategias tienen h(n) = 8 y para el segundo ejemplo A* Manhattan tiene h(n) = 19 y Secuencias h(n) = 10. Por lo que en este aspecto A* Manhattan domina a Secuencias. Tambien podemos ver en la segunda tabla que el coste de la búsqueda es mayor en Secuencias que en A* Manhattan.

3.3 Como podemos ver en las tablas, en el primer ejemplo ambas estrategias tienen el mismo coste de búsqueda, pero en el segundo la estrategia Secuencias tiene un coste mayor. Del estado inicial del primer ejemplo podemos deducir que para la estrategia Secuencias h(n) = 8 y para Descolocadas h(n) = 5, y del segundo ejemplo podemos deducir que para Secuencias h(n) = 10 y para Descolocadas h(n) = 7. Podemos observar que en este caso Secuencias domina a Descolocadas.