Práctica 1: Resolución de un problema mediante búsqueda en un espacio de estados

Sistemas Inteligentes
Grado en Informática Industrial y Robótica
ETSInf - DSIC - UPV

2025/2026

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 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 de trabajo que crees 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`.
4. Pulsa en el botón `'Enter initial state'` e inserta el estado 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`.
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.
9. Para conocer el valor heurístico del estado mostrado, pulsa el botón `'Compute H cost'` (sólo funciona si la estrategia de resolución seleccionada usa una heurística).

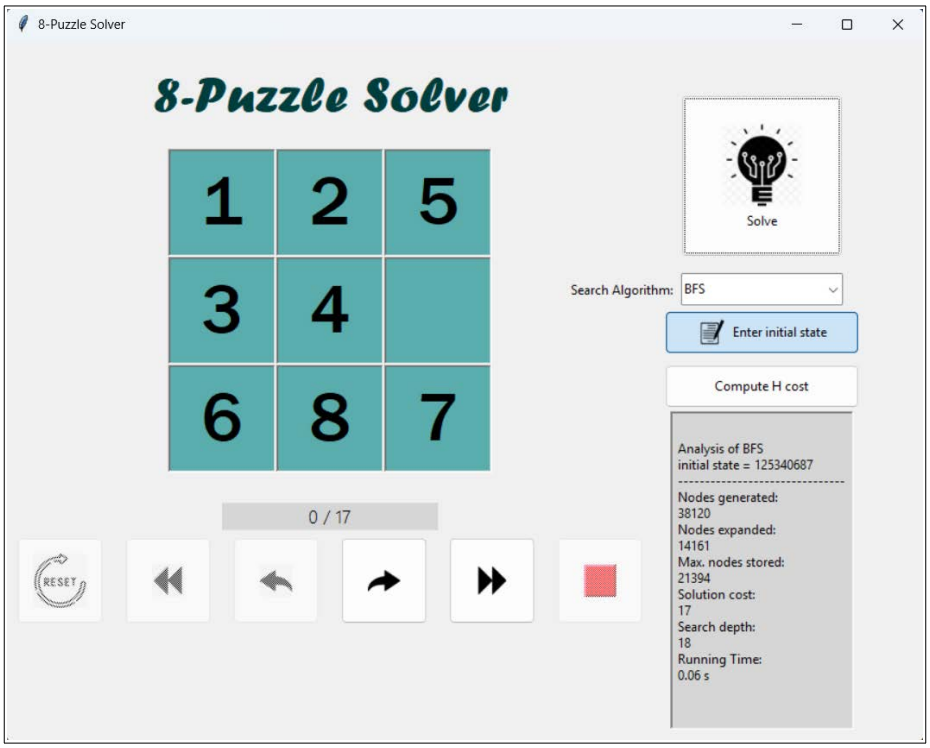

Figura 1: Interfaz principal del programa del puzzle

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. Ocho 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:

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

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, los cuales sólo podrán hacerse cuando el cuadro vacío está en la casilla central. Si el cuadro vacío está en una esquina entonces el número de movimientos válidos es 2, y 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)*, aquí solo se 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 uno.

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 $f(n) = g(n) + h(n)$, con $g(n)$ factor coste correspondiente a la profundidad del nodo n en el árbol de búsqueda y $h(n) = D(n)$ distancia de Manhattan (suma de las distancias en horizontal y vertical de cada ficha del puzzle desde su posición actual a la posición en el objetivo).

A* Euclidean Esta es una búsqueda de **tipo A** que usa la **distancia Euclídea** como función heurística; la expansión de los nodos se realiza según la función $f(n) = g(n) + h(n)$, con $g(n)$ factor de coste correspondiente a la profundidad del nodo n en el árbol de búsqueda y $h(n) = D(n)$ distancia Euclídea, es decir, la suma de las distancias en línea recta de cada ficha del puzzle desde su posición actual a su posición 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*** con límite de la búsqueda determinado por el valor-f, es decir, por la función $f(n) = g(n) + h(n)$, con $g(n)$ factor de coste y $h(n) = D(n)$ distancia de Manhattan. El límite de una iteración i de IDA* es el valor-f más pequeño de cualquier nodo que haya excedido el límite en la iteración $i - 1$.

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 a modificar o añadir. A continuación, se muestra una lista de verificación para comprobar rápidamente si se ha modificado todo el código pertinente. En el resto de esta sección se detalla el código a modificar en cada uno de los tres ficheros Python que componen la aplicación.

Lista de verificación

- `interface.py`
  - Lista desplegable de estrategias (líneas 90-93 aprox.)
  - Función `computeHCost` (línea 177 aprox.)
  - *(Sólo si requiere profundidad máxima)* Función `selectAlgorithm` (línea 274 aprox.)
  - Función `solveState` (línea 422 aprox.)
- `main.py`
  - Función que implementa la nueva heurística (línea 110 aprox.)
- `experiment.py`
  - Diccionario `ALGORITHMS` (línea 74 aprox.)

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. Esto se encuentra en la función `def __init__(self, master=None):` (líneas 90-93 aprox.)

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

- Añadir el código necesario a la función `computeHCost` para que el botón `'Compute H cost'` de la interfaz realice la llamada a la función heurística correspondiente, según la estrategia de búsqueda seleccionada en la lista desplegable (líneas 177 aprox.)

```
def computeHCost(self, event=None):
    global currentState
    algo = self.algorithmbox.get()
    ...
    else:
        if algo in ['A* Manhattan', 'Voraz (Manhattan)', 'IDA* Manhattan']:
            result = f'Manhattan distance: {main.getManhattanDistance(currentState)}'
        elif algo == 'A* Euclidean':
            result =  f'Euclidean distance: {main.getEuclideanDistance(currentState)}'
        else:
            result = 'No heuristic used'
        messagebox.showinfo(message='Heuristic Cost', detail=result)
```

- 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 para solicitar la máxima profundidad. Para ello se modifica la función `selectAlgorithm` (línea 274 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 422 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 `graph Search` 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 `get ManhattanDistance`, `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
   tot = 0                                    # (en forma de cadena texto)
   for i in range(1,9):                       # Recorre las 8 piezas
     goal = end_state.index(str(i))           # Pos. de pieza i en est. final (end_state)
     goalX = int(goal/ 3)                     # Pasa end_state de cad. a matriz de 3×3
     goalY = goal % 3                         # Pieza en fila goalX y columna goalY
     idx = state.index(str(i))                # Misma operación ...
     itemX = int(idx / 3)                     # para la fila idx y columna idy ...
     itemY = idx % 3                          # que ocupa esa pieza en el estado actual
     tot += (abs(goalX - itemX) +             # Cálculo de distancias
             abs(goalY - itemY))
  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 calcular las posiciones de cada pieza en una matriz de 3×3 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 como se observa en la Figura 2.

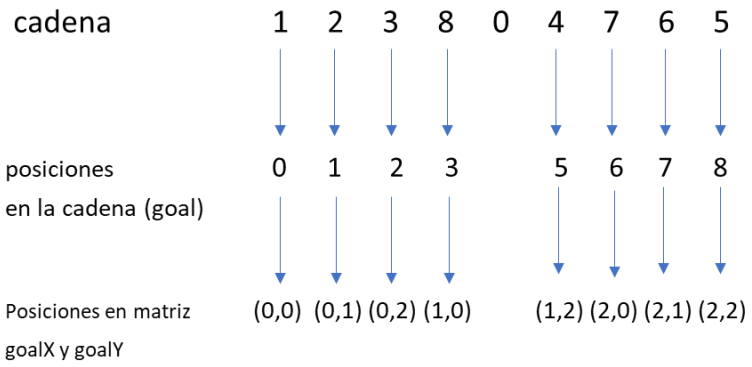

Figura 2: Transformación de cadena representando un estado a la matriz 3×3.

4.3. Fichero `experiment.py`

Este fichero está pensado para ejecutar experimentos que resuelven diversas configuraciones del 8-puzzle con un algoritmo dado, mostrando posteriormente métricas que evalúan dicho algoritmo. Para ejecutar experimentos con una nueva estrategia de búsqueda desarrollada, se debe añadir al diccionario `ALGORITHMS` (línea 64 aprox.) una nueva entrada con sus correspondientes parámetros.

```
ALGORITHMS = {
    'bfs': ('graphSearch', {'function_g': main.function_1, 'function_h': main.function_0}),
    ...
    'a_star_euclidean': ('graphSearch', {'function_g': main.function_1, 'function_h':
                                                        main.getEuclideanDistance})
}
```

5. Trabajo a realizar

La tarea a realizar por cada grupo de prácticas (bien individual o un grupo de dos personas como máximo) sobre el programa del 8-puzzle es la implementación y validación de tres funciones heurísticas y la resolución de una serie de cuestiones con base en una experimentación contrastada.

El día del examen de prácticas todos los estudiantes deben subir a su tarea:

- El código Python operativo completo incluyendo las 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 el mismo fichero de código Python a sus correspondientes tareas (**NOTA: indicad claramente los nombres de las dos personas del grupo en dicho fichero**). El fichero de texto que contiene las respuestas a las preguntas formuladas será el propio de quien realice el examen.

5.1. Implementación de nuevas heurísticas

A continuación se detallan las tres funciones heurísticas a implementar. Una vez implementadas, se debe validar su correcto funcionamiento sobre los estados iniciales definidos en el fichero de texto "Casos de prueba para las heurísticas a implementar", disponible en PoliformaT.

Las funciones heurísticas a implementar son:

1. La función **PIEZAS_DESCOLOCADAS** ($W(n)$), que es el número de casillas en el estado actual que no coinciden con su valor en el estado final (sin incluir al blanco). Con $h(n) = W(n)$ será una búsqueda de tipo A.
2. La función **SECUENCIAS** ($Sec(n)$). Con $h(n) = Sec(n) = 3S(n)$ se hará una búsqueda de tipo A (con $f(n) = g(n) + 3S(n)$).

   $g(n)$: factor coste asociado al nivel de nodo n en el árbol de búsqueda.

   $S(n)$: secuencia obtenida recorriendo sucesivamente las casillas del puzzle, excepto la central, y anotando en la cuenta:

   - 2 puntos por cada ficha no seguida por su ficha sucesora
   - 0 puntos para las otras
   - 1 si hay una ficha en el centro

   Ejemplo de cálculo:

2	8	3
1	6	4
7		5

Para la ficha 2:	Sumar 2 (no va seguida de su ficha sucesora 3)
Para la ficha 8:	Sumar 2 (no va seguida de su ficha sucesora 1)
Para la ficha 3:	Sumar 0 (sí va seguida de su ficha sucesora 4)
Para la ficha 4:	Sumar 0 (sí va seguida de su ficha sucesora 5)
Para la ficha 5:	Sumar 2 (no va seguida de su ficha sucesora 6)
Para la ficha 0:	Sumar 2 (no está en el centro)
Para la ficha 7:	Sumar 2 (no va seguida de su ficha sucesora 8)
Para la ficha 1:	Sumar 0 (va seguida por su ficha sucesora 2)
Para la ficha 6:	Sumar 1 (ficha en posición central).
TOTAL	11 puntos

   Por tanto, $h(n) = Sec(n) = 3S(n) = 33$
3. La función **FILAS_COLUMNAS** ($h(n) = FilCol(n)$), con la que se hará una búsqueda de tipo A. En esta heurística, para cada ficha del tablero:

   - Si la ficha no está en su fila destino correcta se suma 1 punto
   - Si la ficha no está en su columna destino correcta se suma 1 punto.

   Por tanto, el mínimo valor de esta heurística para una ficha es 0 (cuando la ficha está colocada correctamente en su posición objetivo) y el máximo valor es 2 (cuando ni la fila ni la columna de la posición de la ficha son iguales a valor de la fila y columna de la posición objetivo).

5.2. Experimentación

Se provee el script `experiment.py`, que facilita realizar evaluaciones comparativas empíricas de las diferentes heurísticas provistas e implementadas (así como estrategias de búsqueda, si se desea).

Para ello, el script recibe un conjunto de estados iniciales (fichero de texto) y un algoritmo mediante el que resolverlos. Cada estado inicial se soluciona y se anotan sus métricas de evaluación (las mismas que muestra la interfaz). Finalmente, se muestran los valores medios de las métricas, agregando los individuales de cada ejecución. Dichos valores evalúan el algoritmo escogido, mostrando su coste en memoria y tiempo y la calidad de las soluciones encontradas.

El script se puede ejecutar con modo verbosidad y/o modo guardado. El modo verbosidad (opción `--verbose`) informa en tiempo real de la instancia siendo resulta (num/total) y muestra métricas detalladas (media, mediana, mínimo, máximo y desviación estándar). El modo guardado (opción `--save`) genera, al final de la ejecución, dos ficheros CSV: "`details_<algoritmo>_<timestamp>.csv`" y "`summary_<algoritmo>_<timestamp>.csv`", siendo `<algoritmo>` el algoritmo seleccionado y `<timestamp>` la hora de finalización del experimento (evita que se sobreescriban resultados). El archivo de detalles (*details*) muestra las métricas individuales para cada instancia resuelta, mientras que el resumen (*summary*) guarda la misma información agregada que se muestra en la terminal. La Figura 3 muestra un ejemplo de salida del script.

```
Loaded 10 valid puzzles from .\data\10_puzzles.txt
Evaluating bfs on 10 puzzles...
Solving puzzle 1/10: 421786053 (solved in 0.0752s)
Solving puzzle 2/10: 173602584 (solved in 0.0420s)
Solving puzzle 3/10: 254701683 (solved in 0.3867s)
Solving puzzle 4/10: 430786215 (solved in 0.3390s)
Solving puzzle 5/10: 061854237 (solved in 0.1662s)
Solving puzzle 6/10: 643801275 (solved in 0.3766s)
Solving puzzle 7/10: 346257081 (solved in 0.1782s)
Solving puzzle 8/10: 687304152 (solved in 0.3877s)
Solving puzzle 9/10: 046187325 (solved in 0.5614s)
Solving puzzle 10/10: 253641870 (solved in 0.3415s)

============================================================
PERFORMANCE EVALUATION RESULTS
============================================================
Algorithm: bfs
Total puzzles: 10
Solved puzzles: 10
Success rate: 100.0%

PERFORMANCE METRICS       MEAN         MEDIAN       MIN          MAX          STD DEV
--------------------------------------------------------------------------------------
Nodes Generated           174151.40    206605       27593        325499       94286.98
Nodes Expanded            64720.70     76767        10252        121119       35088.71
Max Nodes Stored          83522.60     99776        15684        143661       41240.11
Solution Cost             20.80        22           16           24           2.35
Max Depth                 21.80        23           17           25           2.35
Execution Time (s)        0.2855       0.3403       0.0420       0.5614       0.1634
```

Figura 3: Ejemplo de salida del script `experiment.py` en modo verbosidad

Preguntas a resolver. Dado el fichero de casos de prueba y los conjuntos de entrada de la carpeta "`data`", realiza las experimentaciones que consideres necesarias para responder razonadamente a las siguientes preguntas:

1. La estrategia de búsqueda implementada con la función heurística Secuencias, ¿es un algoritmo A*? Justifica la respuesta.
2. La estrategia de búsqueda implementada con la función heurística Filas_Columnas, ¿es un algoritmo A*? Justifica la respuesta.

3. 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).

4. Compara las estrategias de búsqueda implementadas con las heurísticas Piezas_Descolocadas y Filas_Columnas e indica cuál de las dos estrategias devuelve mejores soluciones (calidad de la solución y coste de la búsqueda).

Para responder se recomienda ejecutar varias configuraciones del puzzle y anotar los resultados de todas las estrategias de búsqueda que se muestran en el programa siguiendo el modelo de tabla de la última página (Cuadro 1), además de para las tres nuevas funciones heurísticas implementadas (Piezas_Descolocadas, Secuencias y Filas_Columnas).

	BFS	DFS (GS)	DFS (Backtr)	Voraz	ID
Nodes generated					
Nodes expanded					
Max nodes stored					
Solution cost					
Solution depth					
Runnning time					

	A* Manhattan	A* Euclídea	IDA* Manhattan
Nodes generated			
Nodes expanded			
Max nodes stored			
Solution cost			
Solution depth			
Runnning time			

	Descolocadas	Secuencia	Filas_Columnas
Nodes generated			
Nodes expanded			
Max nodes stored			
Solution cost			
Solution depth			
Runnning time			

Cuadro 1: Ejemplo de tabla de recogida de resultados