Entrega 2°: Búsqueda Heurística - Zen Puzzle Garden

Hans Schaa

Jun 2023

1 Instrucciones de uso proyecto c++

El repositorio para descargar el proyecto es: `https://github.com/hansschaa/ZenGardenSolver`. Hacer *git clone* y luego *make*. El proyecto c++ usa la librería boost, por ende es requisito tenerla instalada, hacer `sudo apt-get install libboost-all-dev` para su instalación.

El proyecto acepta parámetros por consola para su ejecución. Son en total 6 argumentos, la línea tiene la estructura:

```
./ejecutable boards/6x6/tablero.txt a b c d e
```

Los argumentos anteriores se describen a continuación:

- a: Especifica si juega una persona (0) o la IA (1).
- b: Especifica el tipo de algoritmo de búsqueda a utilizar.
    1. Breadth First Search (BFS).
    2. A*.
    3. IDA*.
- c: 1 para mostrar ruta o 0 para no mostrar.
- d: Este parámetro representa la dimensión del tablero a evaluar, el proyecto tiene tableros de 4x4, 6x6 y 8x8, se debe señalar la dimensión del tablero a jugar. Se limitó el proyecto c++ a tableros cuadrados.
- e: Límite en milisegundos para la condición de término de IDA*.

Un ejemplo es el siguiente:

```
./tablut boards/6x6/board1.txt 1 2 1 6 10000
```

Donde jugará la IA con el algoritmo de búsqueda IDA*, mostrará la ruta encontrada si es que hay solución, 6 es la dimensión del tablero y por último hay un límite de 10000 milisegundos.

Figure 1: Ejemplo de una secuencia de 2 movimientos para el monje. Primer movimiento hacia la izquierda y luego hacia abajo.

2 Descripción del juego

Zen Puzle Garden es un puzzle geométrico contextualizado en los jardines rocosos japoneses. La meta de cada uno de los tableros es rastrillar toda la superficie sin romper las líneas rastrilladas. Los niveles de juego pueden variar desde tableros pequeños de 4 x 4 mosaicos hasta algunos más grandes de 12 x 12 mosaicos. También existen tableros con formas más complejas, donde los tableros no son cuadrados ni rectangulares.

El movimiento del monje es uno a la vez, es decir que el usuario señala en qué dirección desea mover al personaje y este toma esa ruta para rastrillar la superficie. Una secuencia de 2 movimientos sería el que se puede apreciar en la figura 1, donde el monje se mueve en el eje -x, colisiona con una roca y debe bajar en el eje -y.

Cómo se pudo apreciar el tablero tiene distintos tipos de obstáculos que dificultan la resolución del nivel. Una vez toda el área esté rastrillada, el nivel se completa.

2.1 ¿Cómo jugar en el proyecto c++?

El proyecto permite que el usuario juegue un tablero dado, para ello solo asegúrese de indicar 0 en la opción para el modo de juego. Un ejemplo es la siguiente instrucción para iniciar una partida:

```
./tablut boards/6x6/board1.txt 0 0 0 0 0
```

Se presentará un tablero enumerando las filas y columnas como se aprecia en la figura 2. Ingresa el ID de la columna o fila por donde desees que empiece a rastrillar el monje. Si este se topa con un obstáculo, debes indicar la dirección (i,j) para que el monje tome esa dirección.

3 Breve descripción de las técnicas implementadas

A continuación, se describen algunas características del proyecto junto a los métodos de búsqueda implementados:

Representación Se usaron 2 `boost::dynamic_bitset` para representar cada figura del juego, estas son las siguientes:

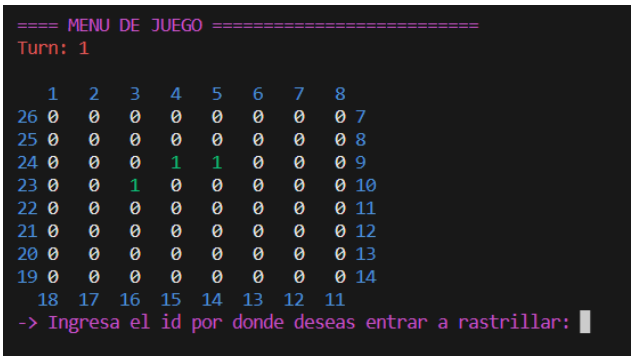


Figure 2: Tablero enumerando filas y columnas para juego sin IA

1. Obstáculos o área rastrillada: En términos prácticos estos elementos tienen la misma función dentro del juego (obstaculizar o delimitar el movimiento del monje). Son representados por 1s verdes en la salida estándar,
2. Monje: Cómo se mencionó, el monje en ocasiones queda dentro del tablero por culpa de algún obstáculo. Se representa por medio de un 1 rojo.

BFS Se implementó BFS como algoritmo de búsqueda en el árbol de estados que se genera al jugar Zen Puzzle. Este comienza con un nodo inicial o raíz y explora todos los estados sucesores para encontrar si alguno de ellos es la solución. Si no se encuentra la solución, se repite el paso de búsqueda en los sucesores. Los nodos se van visitando de forma gradual hasta encontrar un estado que soluciona el tablero.

A* Al igual que el algoritmo anterior, A* es otro algoritmo de búsqueda en árboles. A* es una mejora del algoritmo BFS al incluir en su proceso una heurística que estima a cuanto esta un nodo en específico de alcanzar el estado solución. La función de costo total f para un nodo x se calcula como la suma del valor de la profundidad g en el árbol donde se encuentra el nodo x, y el valor resultante de la heurística h:

$$f(x) = g + h$$

En esta ecuación, g representa la profundidad donde se encuentra el nodo x en el árbol. Por otro lado, h es el valor estimado de la heurística, que proporciona una estimación del costo restante desde el nodo x hasta el nodo objetivo.

Al sumar g y h, la función f combina el costo acumulado desde el nodo inicial hasta el nodo actual, y la estimación heurística del costo restante. Esto permite al algoritmo A* priorizar los nodos con un menor valor de f, lo que tiende a guiar la búsqueda hacia la dirección más prometedora y acelerar la búsqueda del camino más corto hacia el nodo objetivo.

IDA* Algoritmo de búsqueda sin memoria para encontrar la ruta más corta entre un nodo inicial y un nodo meta en un árbol de estados. Este es una modificación del algoritmo A* que evita almacenar la lista abierta y cerrada en memoria. Al igual que A*, IDA* usa un costo f que es la suma del $g + h$ anteriormente descrito.

IDA* opera por medio de una búsqueda en profundidad restringida por un límite inicial que va cambiando en la ejecución. Si el objetivo no se encuentra dentro del límite, se incrementa el límite de costo al valor mínimo de los costos de los nodos que superaron el límite en la búsqueda anterior. Al igual que A*, IDA* garantiza encontrar la solución óptima si la función heurística es admisible.

Heurística La Heurística utilizada se resumen en contar la menor cantidad de líneas horizontales o verticales que caen sobre los espacios libres del tablero.

4 Análisis y resultados

Para llevar a cabo las pruebas se dispone de 3 grupos de tableros: 4x4, 6x6 y 8x8, donde cada grupo tiene 3 tableros diferentes: fáciles, normales y difíciles. Esta categorización fue realizada tomando en cuenta el tiempo que tarda el algoritmo en promedio para solucionar el tablero, estos son ordenados de forma ascendente. Se midió tiempo de ejecución, largo de la ruta resultante y cantidad de nodos explorados.

4.1 Tiempo de ejecución

Los tiempos de ejecución fueron obtenidos ponderando los resultados de 3 intentos. A continuación, se muestran los gráficos divididos por dimensionalidad del tablero: 4x4 (3(a)), 6x6 (3(b)) y 8x8 (3(c)). De las figuras podemos concluir lo siguiente:

1. De la figura 3(a), en promedio BFS tiene un rendimiento intermedio para tableros pequeños. A* es el que tiene mejores tiempos, IDA* sólo es el superior en el tablero 4x4:board1, esto porque la búsqueda en profundidad para un tablero sin obstáculos se encuentra en las primeras llamadas recursivas.

2. Para la figura 3(b), la tendencia cambió un poco dejando a IDA* como el algoritmo mas lento, A* sigue siendo el que tiene mejores tiempos. El retardo de IDA* puede deberse al recalculo de la función heurística para cada iteración a partir de la raíz, también puede ser que el revisar en profundidad para este tipo de puzles no sea muy buena idea cuando se trata de velocidad, mas no de memoria (en el próximo ítem hay un ejemplo de esta idea).

3. Por último en la figura 3(c) podemos ver que BFS sólo resolvió el tablero 8x8: board1. En este caso IDA* tiene mejores resultados que BFS tan solo por solucionar todos los tableros de 8x8. Son curiosos los 2.450.310 milisegundos (41 minutos) que le tomó a IDA* resolver este tablero, IDA* tiene un alto potencial para árboles profundos. A* tiene un excelente rendimiento, dejando muy por debajo a los demás algoritmos.

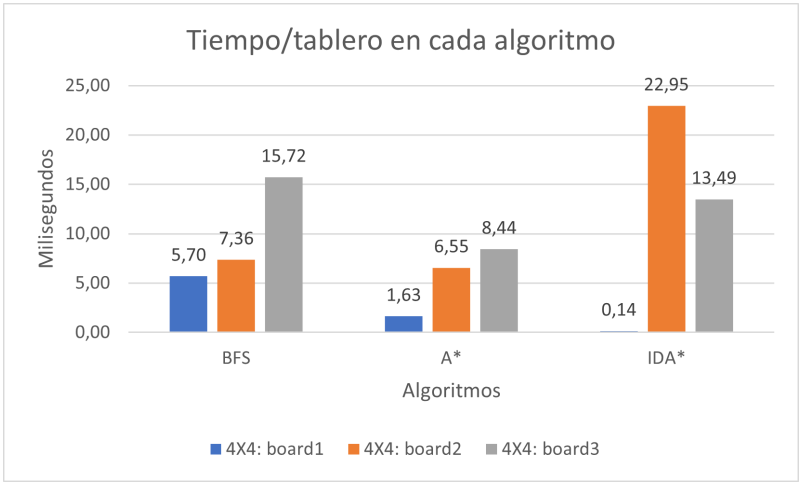
4.2 Total de nodos explorados y largo de la ruta

El largo de solución encontrada para todos los algoritmos y tableros es el mismo, a excepción de los tableros 8x8-board2 y 8x8-board3 donde BFS no llegó a la solución lanzando la sentencia `abort`. Creo que esto se debe a un uso excesivo de recursos. Respecto de los nodos revisados podemos mencionar lo siguiente:

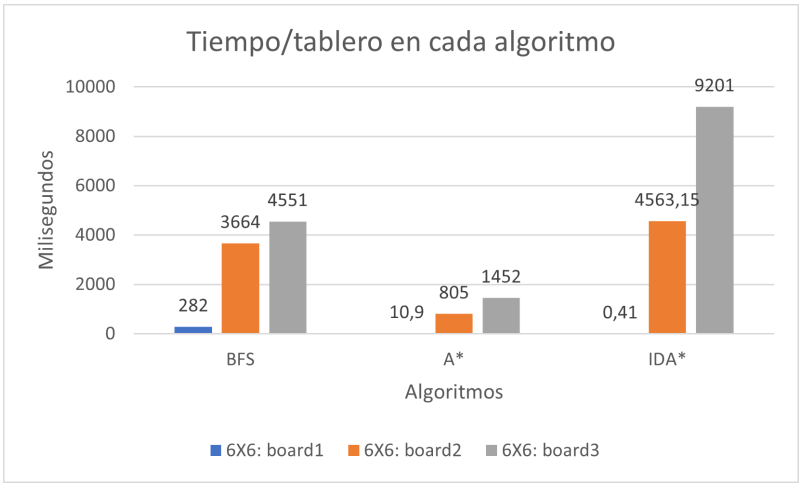
1. De la figura 4(a) se aprecia que existe una correlación positiva entre el aumento de las dimensiones de los tableros frente a los nodos visitados. Esta tendencia se repite en los demás algoritmos.

2. La figura 4(b) es la que presenta mejor rendimiento respecto de los nodos visitados, esto se debe a su heurística para encontrar la solución revisando solo nodos prometedores.

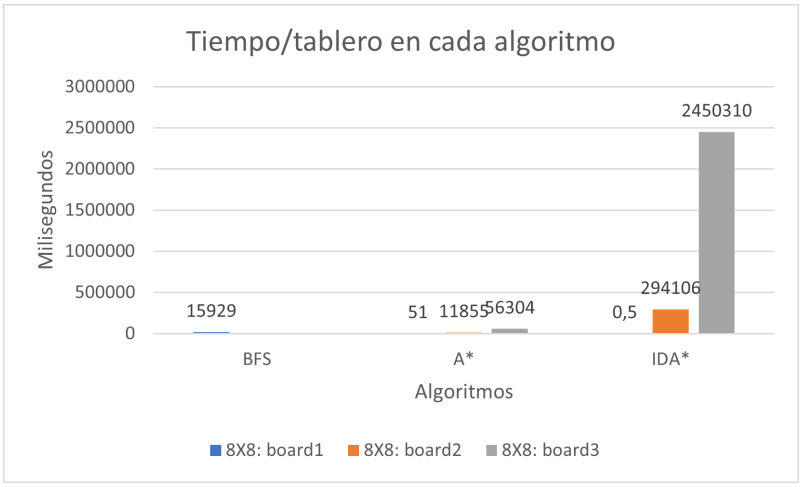
3. En la figura 4(c) se aprecia que a excepción de los tableros board1 para cada dimensión, este explora una gran cantidad de nodos. Esto debe ser por la naturaleza iterativa del algoritmo lo que se podría solucionar con una tabla de transposición.



(a)

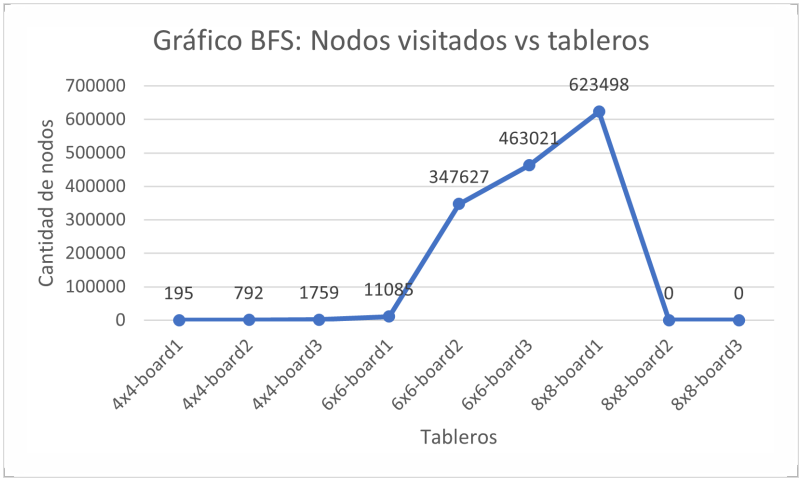


(b)

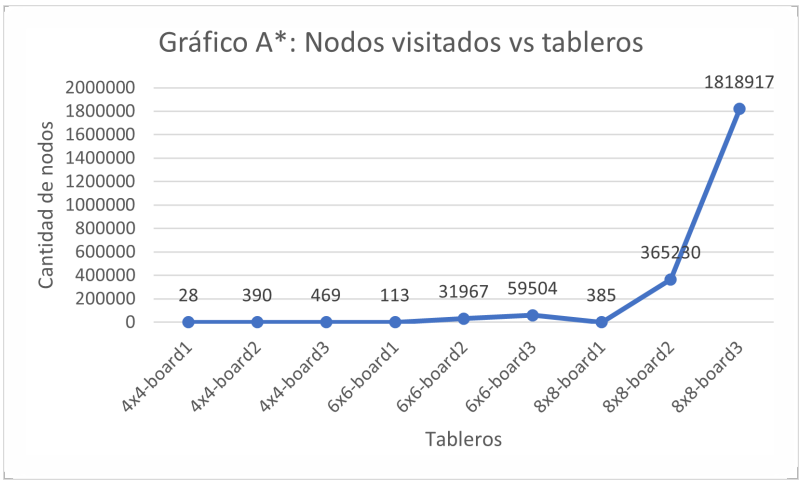


(c)

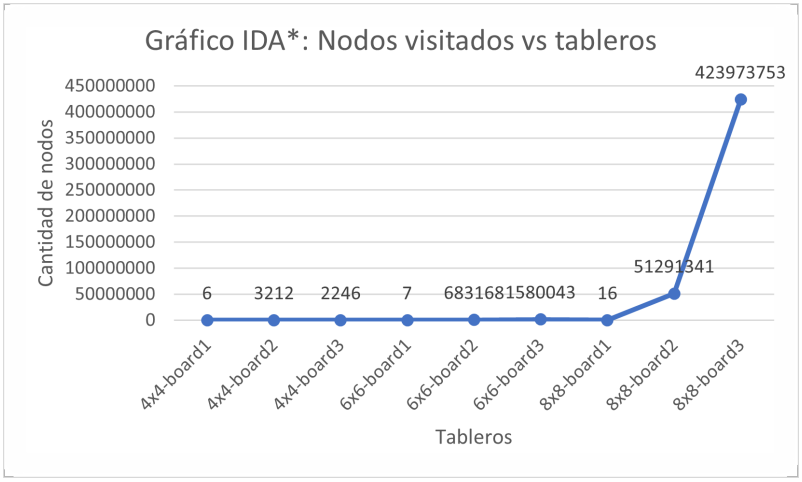
Figure 3: (a) Milisegundos tomados por cada algoritmo en la solución de tableros de 4x4. (b) Milisegundos tomados por cada algoritmo en la solución de tableros de 6x6. (c) Milisegundos tomados por cada algoritmo en la solución de tableros de 8x8.



(a)



(b)



(c)

Figure 4: (a) Nodos revisados para la solución de tableros de 4x4. (b) Nodos revisados para la solución de tableros de 6x6. (c) Nodos revisados para la solución de tableros de 8x8.