Sistemas de Inteligencia Artificial

Trabajo Práctico Especial 1

Juego: 0h-n0

Profesor:

María Cristina Parpaglione

Alumnos:

Francis Gilly	54053
Lucas Kania	54257
Emilio Tylson	54022

Intoducción

El siguiente trabajo muestra diferentes formas de encontrar la solución del juego 0hn0 bajo diferentes algoritmos de búsqueda, ya sean algoritmos desinformados de la lógica del juego como DFS, BFS y IDDFS; como algoritmos informados A* y Greedy. Luego se mostrará la performance de los algoritmos junto con una comparación entre estos.

Descripción del juego

El juego consta de un tablero donde se puede insertar *tokens.* Estos *tokens* pueden ser de tipo *floor* o *wall* (Figura 1.a y 1.c respectivamente). La diferencia entre estos dos es que los *token floor* pueden verse entre ellos siempre en cuanto estén contiguos en la fila o columna y no haya un *token wall* entre ellos (Figura 2.a y 2.b). Es decir que un *token floor* no puede ver a través de un *token wall.* Dentro de los *token floor* existen los *token* con *label* (Figura 1.b), en el cual se indica la cantidad de *token floor* contiguos que debe "ver" en su fila o columna, sin la interrupción de un *token wall*. A estos *token* contiguos que pueden ver el *token label* se los define como **"vecinos"** a lo largo del informe.
Definido esto, el tablero inicial (Figura 3.a) consta de un tablero en el que están puestos todos los *token label* y ciertos *token wall*. El objetivo es llenar los espacios vacíos con *token floor o wall,* de forma tal que todos los *token label* vean tantos *token floor* como indica su *label*; y que no quede ningún *token floor* aislado, es decir sin ser visto por ningún *token label.*

Descripción general de la búsqueda de la solución

La búsqueda de la solución para el juego se dividió en dos subproblemas por motivos de eficiencia que a continuación serán explicados.
El primer subproblema se encarga de buscar "satisfacer" todos los tokens con *label,* es decir que vean tantos vecinos como indica su *label*. A pesar de que el estado solución aparente ser la solución definitiva al juego, esta búsqueda no contempla *tokens floor* que estén aislados en el estado solución, es decir que en cualquiera de las cuatro direcciones no ven a ningún *token* con *label*, ya sea por que hay *walls* o por las limitaciones del tablero. Para afrontar este primer subproblema se decidió crear un estado inicial en el cuál todos los espacios vacíos del tablero se llenan con *token floor*, con lo cual las reglas se reducen a insertar *token wall* en lugares posibles (no sobre un *token* con label, ni sobre otro *wall*). Estas reglas de inserción de *walls* son generadas tomando cada *token label* y viendo a partir de sus cuatro direcciones posibles en este juego (arriba, abajo, izquierda y derecha) todos los lugares posibles donde insertar un *token wall*. Cabe destacar que generando estas reglas puede dejar puntos ciegos donde nunca se considera meter un *wall* debido a que en esas posiciones no afectan a ningún *token label*.
Como se mencionó en el párrafo anterior, el estado *goal* del primer subproblema puede dejar *token floor* aislados, por lo que el segundo subproblema se encarga de solucionarlo dejando un estado *goal* que es solución definitiva del juego. El estado inicial de este subproblema es el estado *goal* del primer subproblema. El set de reglas en este caso consiste en considerar todos los espacios donde se puede colocar un *wall*, y en

consecuencia es un *set* de reglas más reducido (al partir de un estado más cercano a la solución final).
Si bien se puede resolver todo en un único problema, esto implicaría que el set de reglas tenga en cuenta los espacios ciegos que se ignoraron al realizar el subproblema uno y en consecuencia tener un mayor *branching factor* (mayor cantidad de reglas aplicables) que repercute a la hora de expandir nodos. Por otro lado en los algoritmos de búsqueda informada, y debido a las heurísticas desarrolladas en este trabajo, realizar todo en un único problema implicaría aplicar la heurística adecuada a cada estado dependiendo si tiene todos sus *token label* satisfechos.

Búsqueda desinformada

DFS

Para implementar DFS, se usó un *comparator* que ordene de mayor a menor por profundidad, de esa forma quedan primeros en la cola de prioridades los nodos recién accedidos.

BFS

Para realizar BFS, el *comparator* utilizado ordena de menor a mayor por profundidad de forma que los nodos más recientes se agreguen al final de la cola de nodos frontera.

Iterative Deepening Depth First Search

Este método consiste en realizar reiteradas veces DFS hasta un límite de profundidad que se aumenta luego de cada iteración hasta encontrar el estado solución. Para implementarlo, los nodos deben almacenar la profundidad en la que se encuentran. Al tomar los nodos de la frontera se evalúa si se encuentran en la profundidad límite y en ese caso no se expanden.

Búsqueda informada

Se procede a realizar la búsqueda del estado *goal* introduciendo lógica del juego a la búsqueda mediante una función heurística y el costo. A continuación se detallara dichas funciones.

Función de costo

El objetivo del juego es encontrar una disposición final de *tokens floor* y *tokens wall* por lo que no influye en qué orden se colocan los *wall*. Es decir no importa en una primera instancia el costo del camino a la solución, sólo importa la solución. El costo más adecuado para la resolución de este problema es contar la cantidad de *tokens* wall puesta en cada expansión de nodos, es decir que el costo para pasar de cualquier estado a otro, de la forma que se realizaron las reglas es siempre uno, ya que se pone un solo *wall*. De este

modo el algoritmo A* estaría encontrando la solución óptima en costo (siempre y cuando se utilice una heurística admisible), es decir la solución que se tenga que llegar insertando menor cantidad de *walls*.

Heurística no admisible - H1

La función de esta heurística es determinar cuán bueno es un tablero en función a los *tokens* que poseen más vecinos de lo que dicta su *label*. Para este propósito se define una función “weight” que se encarga de determinar la cantidad de vecinos visibles por un *token* con label y le resta el valor del *label*. Esta función estaría determinando cuántos vecinos demás ve cada token. En consecuencia se puede observar casos en que el *token* ve menos vecinos que lo que dice su *label*, claramente estas situaciones son estados en el tablero que no llevan a ninguna solución. Por este motivo se quiere que una vez que el estado sea puesto en la frontera no sea elegido para expandir por ningún algoritmo informado, para eso se necesita que la función en vez de dar un número negativo, entregue un valor demasiado alto para que en lo posible no sea analizado, en este caso lo que devuelve la función es la representación del número entero grande que nunca podría alcanzarse en el peor de los casos.
La función heurística final se encarga de realizar la sumatoria de la función “*weight”* anteriormente descrita aplicada a cada *token* con label. Se puede observar que esta función aplicada en el estado *goal* en el cual cada *token* con label está satisfecho (cada *token* ve tantos vecinos como dicta su *label*) da 0. En cambio, si se llega a un estado en el que al menos un *token* ve menos vecinos que su *label*, la heurística da un valor muy grande porque no tiene sentido expandir ese nodo. La heurística tiene un valor decreciente a medida que más vecinos son satisfechos, lo que va guiando la búsqueda informada hacia mejores estados (más *tokens* satisfechos), y en consecuencia hasta el estado *goal*.
Dado que la función de costo es la cantidad de *walls* insertadas, esta heurística no es admisible puesto que no estima la cantidad de *walls* que se necesitan poner para llegar al estado *goal*, sino da un parámetro de cuán bueno es el estado. Es decir que un *token* que ve tres vecinos más de lo que dice su *label* no indica que se requieren insertar tres walls para satisfacerlo.
Dado que esta heurística no es admisible pero da un claro parámetro de la “calidad” del estado esta heurística tiene sentido a ser usada en el algoritmo greedy.

Heurística utilizada en el segundo subproblema

El segundo subproblema presenta una situación más simple de resolver, encontrar un estado *goal* donde no tenga ningún *token floor* aislado. En este caso se utilizó como heurística la cantidad de *tokens floor* aislados. En este caso la heurística es admisible, en realidad es prácticamente h*(n) (costo real del estado del nodo n al goal) pues se está cuantificando la cantidad exacta de *tokens wall* a insertar para llegar al estado *goal,* donde se llega reemplazando los *tokens floor* por *walls.*

Heurística admisible - H2

El objetivo de esta heurística es estimar la mínima cantidad de *walls* que le faltan al tablero para alcanzar el estado solución del primer subproblema.
Una estimación posible de este valor es considerar la cantidad de *tokens* con *label* no satisfechos y considerar que es necesario como mínimo colocar un *wall* por cada uno de ellos para llegar a la solución. Sin embargo, es posible que si dos *tokens* con *label* se encuentran en la misma fila o columna y todavía no existe un *wall* entre ellos, colocando dicho *wall* se logre satisfacer a ambos *tokens* con *label*, con lo cual se estaría sobreestimando y por lo tanto, la heurística no sería admisible.
Para obtener este valor sin realizar una sobreestimación, se requiere reducir el conjunto de *tokens* con *label* de forma tal que se considere sólo un grupo que no se pueda ver entre sí, lo cual, para ejemplificar, es equivalente a la disposición que se desea obtener en el juego “ocho reinas”, es decir que ninguno quede en el campo de visión del otro. Aún bajo esta disposición, puede ocurrir que un *wall* resuelva dos *tokens* con *label* a la vez (al de la fila y al de la columna, de ese *wall*). Por lo que, para subestimar se debe considerar la mitad de la cantidad de *tokens* con *label* que conforman el grupo, ya que es una cota inferior al número de *walls* requeridos para alcanzar el estado solución. Ese resultado es el valor heurístico que se le otorga al estado.
Para la realización del cálculo de dicho valor, se toman los *tokens* con *label* que todavía no alcanzaron a ver la cantidad de *tokens floor* que los satisface y se los coloca en un conjunto en pares *token* - dirección, donde la dirección es hacia abajo o hacia la derecha, para reducir los casos en los que dos *tokens* con *label* se puedan ver simultáneamente. En este momento, si alguno de los *tokens* con *label* ve menos *tokens* de los que requiere, se retorna el valor máximo (el tamaño del tablero, ya que es una cota superior de la máxima cantidad de *walls*), con la finalidad de que el estado no sea expandido, ya que el mismo no conduce a la solución (porque solo se agregan *walls*). Luego, se itera sobre el conjunto, y por cada *token* se recorre el tablero a partir de su posición y según la dirección que conforma el par extraído. El recorrido continúa mientras ese *token* pueda seguir viendo *tokens*, es decir, no se alcance un *wall* ni los bordes del tablero. Durante el recorrido, si se encuentra otro *token* con *label* se elimina del conjunto el par con este último y la dirección en que fue encontrado. De esta forma, se reduce el conjunto a algunos *tokens* con *label* de manera que ninguno de los que se encuentre en éste vea a otro *token* con *label* en la misma dirección.
El tamaño del conjunto dividido dos (porque se habían considerado dos direcciones por cada token y se dijo que la estimación es un *wall* por *token*) es una aproximación a una mínima cantidad de *walls* requeridos para solucionar el tablero. Sin embargo, como se dijo antes, existen casos en los que se estaría sobreestimando. Un ejemplo es el caso de la figura 3, que por la disposición de los *tokens* (Figura 3.a), cada *wall* satisface dos *tokens* con *label* que no se pueden ver entre sí. Entonces, la estimación realizada da que se requieren ocho pares token - dirección que dividido dos, da cuatro *walls* (Figura 3.b), pero con dos *walls* se resuelve (Figura 3.c), por lo que dividiendo por cuatro se logra subestimar estos casos

A*

El algoritmo de búsqueda A* busca la solución de menor costo, en este problema y al tener una única solución, el algoritmo simplemente busca la solución. La función f que utiliza el algoritmo es la suma de la función costo descrita anteriormente, cantidad de *tokens wall*

insertada en cada paso, y la heurística H2 puesto que esta heurística es admisible puesto que subestima la cantidad de *tokens wall* a poner. Es necesario aclarar que este algoritmo busca el camino de menor costo al estado solución, sin embargo en este juego no interesa la cantidad de *walls* a poner, o el camino a la solución; sino, lo único que interesa es la disposición de las piezas en el estado *goal*. Sin embargo a pesar de ésto, el algoritmo da óptimos resultados encontrando el estado solución sin que se acabe la memoria en casi todos los tableros a excepción de: "5x5_1", "6x6_2","6x6_3", "6x6_4" (archivos de la carpeta “doc/board”). Es más, este algoritmo, es el único que pudo resolver todos los tableros del conjunto de *test*.

Greedy

El siguiente algoritmo expande los nodos de la frontera cuyo valor heurístico sea menor. En este caso se decidió usar H1, la heurística no admisible, pues ofrece cuantificar qué tan bueno es un estado a partir del desfasaje de los *tokens label*. A diferencia del algoritmo A* no interviene el costo del camino, sino, sólo la heurística, por lo que el objetivo de este algoritmo es más apropiado a los objetivos del juego ya que es encontrar simplemente el estado *goal* sin que intervenga la cantidad de *walls* puestos. Sin embargo este algoritmo no pudo resolver ciertos tableros: "5x5_1", "6x6_4", "7x7_2", "7x7_4" (se pueden encontrar en la carpeta “doc/board” del proyecto), debido a falta de memoria. Esto se debe a que los tableros expuestos tienen una profundidad de solución muy grande (requieren muchos walls a insertar), por lo tanto el algoritmo puede tomar caminos donde sucesivamente el valor de H1 baja, y darse cuenta que no tiene sentido seguir (a partir de un valor heurístico grande) en una profundidad alta, por lo que tiene que retomar el nodo de menor heurística de los nodo frontera que posiblemente esté a una menor altura.

Resultados

BFS, DFS, IDDFS

BFS resuelve los tableros de 4x4 y uno de 5x5, con profundidades máximas de 5 y 4 respectivamente. Para el resto de los tableros se queda sin memoria. En el caso de DFS, resuelve los mismos tableros que BFS. La performance de ambos depende del tablero pero no depende ni del tamaño ni de la profundidad. En el caso de IDDFS, resuelve tableros de 4x4, tres de 5x5 y uno de 6x6, con profundidades máximas de 5, 7 y 7 respectivamente. Para los demás tableros se queda sin memoria luego del nivel de profundidad 7.

A* vs Greedy

A partir de los gráficos Figura 4 y Figura 5, se puede apreciar que el algoritmo greedy es más eficiente en tiempo y en memoria, pues expande menor cantidad de nodos. Como se encargó de señalar en el informe, esto se debe a la naturaleza del juego pues sólo interesa la disposición final del estado solución y no interesa encontrar el camino óptimo. Es por eso que el algoritmo greedy al no importarle el camino a la solución y usar una heurística que cuantifica la calidad del tablero llega más rápido que A*, que busca el camino óptimo no trascendente para este juego.

Anexo

Tokens:

Figura 1.a *Token floor*

Figura 1.b *Token label*

Figura 1.c *Token wall*

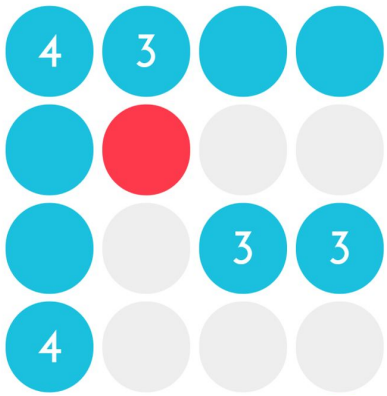

Figura 2.a el *token label* 4 por ejemplo ve tres vecinos a su izquierda y tres vecinos en direccion de abajo

Figura 2.b El token label 1 no puede ver al *token floor* debido a que hay un *wall* en el medio.

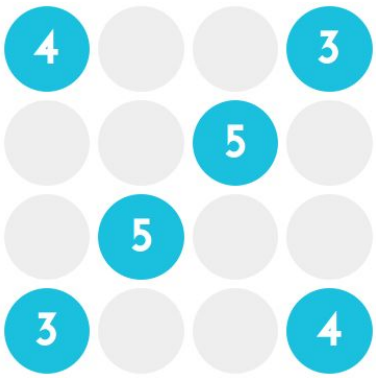

Figura 3.a. Tablero inicial.

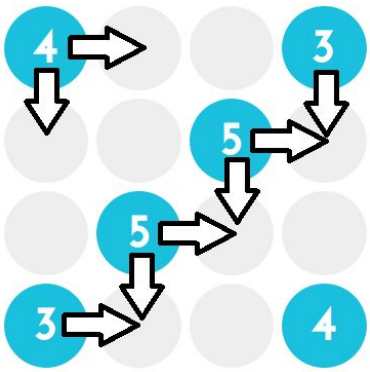

Figura 3.b. *Tokens* con sus direcciones. Se muestran los ocho pares token - dirección almacenados en el conjunto al final del método.

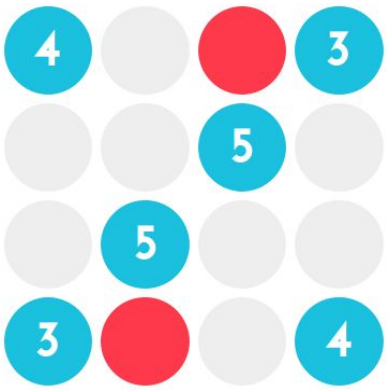

Figura 3.c. Tablero final resuelto con dos *walls*.

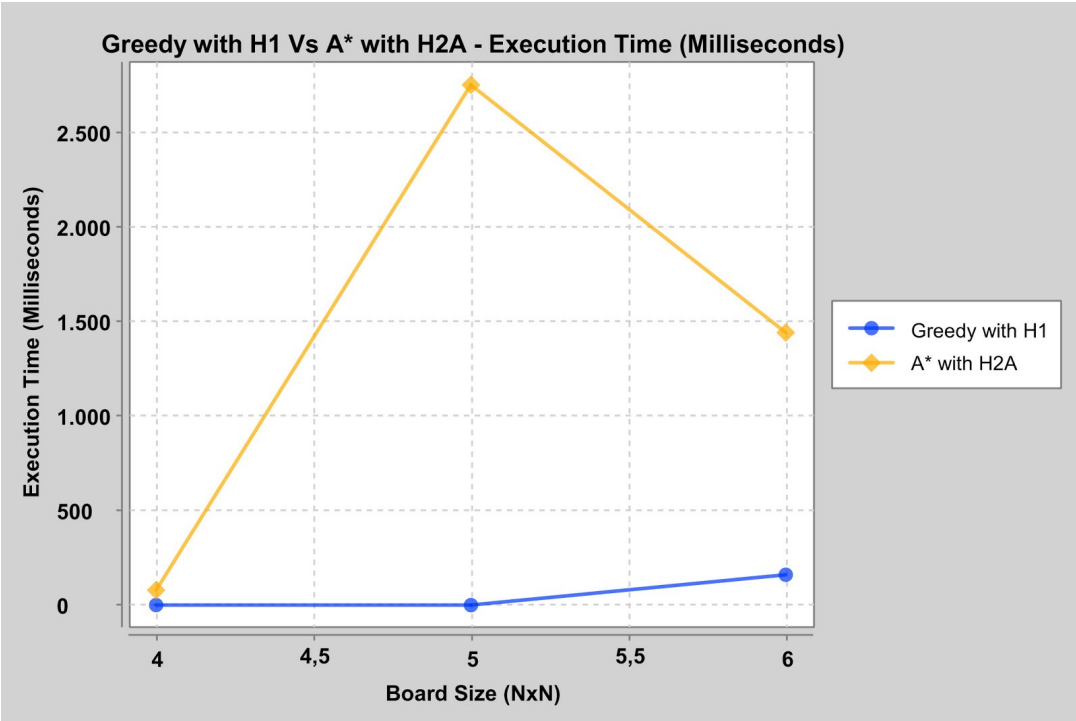

Figura 4. El gráfico compara el tiempo de ejecución promedio para distintos tamaños de tablero. El gráfico color azul corresponde al algoritmo greedy con la heurística H1. El gráfico color amarillo corresponde al algoritmo A* con heurística H2 admisible.

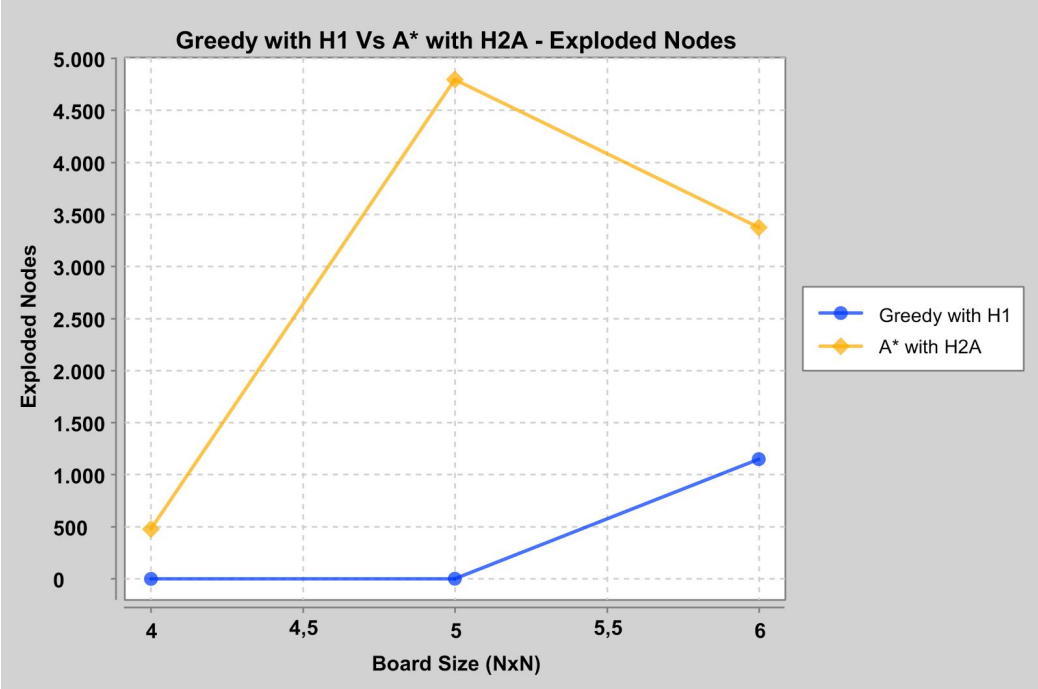

Figura 5. El gráfico compara los nodos expandidos en total para distintos tamaños de tablero. El gráfico color azul corresponde al algoritmo greedy con la heurística H1. El gráfico color amarillo corresponde al algoritmo A* con heurística H2 admisible.