# Análisis de la solución de *Chain Reaction*

Sistemas de Inteligencia Artificial
Instituto Tecnológico de Buenos Aires

Garrigó, Mariano 54393 — Raies, Tomás A. 56099 — Saqués, M. Alejo 56047

## Resumen

Se ha implementado un Sistema de Producción para la resolución genérica de problemas de búsqueda, en base a una interfaz previamente consensuada con la Cátedra y el curso.

Dicho sistema provee una serie de mecanismos que afectan la manera en que se insertan y extraen nodos del conjunto de frontera, efectivamente alterando el comportamiento del algoritmo de búsqueda.

Asimismo, en lo que respecta al problema de *Chain Reaction*, se ha implementado una serie de heurísticas a los efectos de acelerar el hallazgo de una solución.

**Palabras clave:** Sistema de Producción, búsqueda desinformada, *breadth-first search*, *depth-first search*, *iterative deepening depth-first search*, *greedy search*, *A* algorithm*, *Chain reaction*, *Hamiltonian path*.

## 1. Búsquedas desinformadas

En esta sección se discutirán los resultados obtenidos utilizando búsquedas desinformadas.

### 1.1. *Depth-first search* y *Breadth-first search*

En las tablas adjuntadas, se puede observar claramente como evoluciona el problema de uso de memoria en BFS en comparación a DFS: para el primer caso, BFS expande una cantidad de nodos dos ordenes de magnitud mayor que DFS. Para el caso de $7X7$, mientras DFS sigue − con dificultades − hallando una solución, BFS provoca un error de falta de *heap* para asignar más memoria.

Sin embargo, DFS muestra sus deficiencias en el caso $8X8$, donde se terminó su ejecución por un *timeout* definido en 10 minutos. Por lo general, DFS provee una mejor *performance* de BFS en casos en los que existe más de una solución. Para el caso de *Chain Reaction*, la cantidad de soluciones está intrínsecamente asociada a la estructura del grafo obtenido a partir del tablero, pudiendo eventualmente ser única.

En casos a analizarse más adelante, se observará que el valor asignado al *timeout* sumamente generoso para el tiempo de ejecución que demandan las implementaciones que utilizan heurísticas.

### 1.2. *Iterative deepening DFS* (IDDFS)

El objetivo de la profundización iterativa es realizar una búsqueda en anchura sin el costo en memoria de un BFS tradicional. Por ende, lo que se busca es realizar una implementación de DFS limitada por un valor máximo de profundidad.

En el problema de *Chain Reaction*, el objetivo es encontrar un camino hamiltoniano sobre los nodos del tablero, utilizando las reglas que aplican al problema. Es decir, se quiere encontrar un camino que recorra los $N$ vértices una

única vez, por ende, el nodo solución estará en la máxima profundidad posible.

Visto esto, para el caso de *Chain Reaction*, como siempre la solución estará en la máxima profundidad, IDDFS es ineficiente a la hora de encontrar el resultado. Efectivamente, la solución provista por IDDFS es equivalente a la de DFS sin límites de profundidad, más el *overhead* de haber ejecutado un DFS limitado por altura para cada una de las alturas no válidas.

# 2. Heurísticas

En esta sección, se discutirán las diferentes heurísticas implementadas, demostrando su admisibilidad.

## 2.1. Heurística sin poda

Esta heurística, a diferencia de las que se exhibirán más adelante, no realiza ningún tipo de poda en base a la situación del tablero que emana del estado a analizar. A continuación sigue la función:

$$h(E) = (Remaining(E)) - \frac{1}{Open(E)} \quad (1)$$

$$Remaining(E) = Total - Occupied(E) \quad (2)$$

Donde $Total$ es la cantidad total de casilleros no vacíos, $Open(E)$ es la cantidad de estados vecinos a $E$. Si $Open(E) = 0 \wedge Remaining(E) \neq 0$, se define $h(E) = +\infty$. Al ser estos estados inalcanzables, un sistema de producción razonable debería evitar insertar casos como estos en el conjunto de nodos frontera, economizándose así el consumo de memoria.

Como podrá verse, esta heurística pondera el *grado* del estado, tomando al tablero como un grafo cuyo nodo actual es el último casillero visitado. Supóngase que, a partir del estado inicial, se expanden 2 nuevos estados. Para $E_1$, $Open(E_1) = 3$, y para $E_2$, $Open(E_2) = 1$. Para ambos, $Remaining(E_1) = Remaining(E_2) = N - 2$, dado que siempre el casillero inicial está ocupado, y al ser la primera expansión, se ha ocupado un casillero adicional. Por ende, $h(E_1) = N-2-\frac{1}{3}$ y $h(E_2) = N-2-1 = N-3$. Entonces, $h(E_2) < h(E_1)$, por lo que, a igual costo de todas las aristas, un algoritmo $A*$ seleccionará $E_2$ como siguiente nodo a expandir. Idéntico será el caso para el algoritmo *Greedy*.

El objetivo que se persigue al priorizar los estados con menor cantidad de caminos abiertos es, o bien fallar rápido − tomando un camino que hace que el siguiente estado sea una hoja en el árbol de búsqueda −, o bien privilegiar recorrer caminos largos en subgrafos poco conexos, con las esperanzas de que los subgrafos no explorados sean lo suficientemente conexos como para poder recorrerlos sin dificultades.

### 2.1.1. Admisibilidad

Definiendo al costo de recorrer un eje en 1, el costo total de la solución es $N - 1$, siendo $N$ el número de casilleros no vacíos. Por ende, a lo sumo $Remaining(E) = N - 1$, dado que siempre el nodo inicial está ocupado, y en el peor caso el casillero actual de $E$ está conectado a todos los otros casilleros en las direcciones válidas, entonces $Open(E) = Rows + Cols - 2$, por lo que como máximo $h(E) = N - 1 - \frac{1}{Rows+Cols-2} \leq N - 1$,

En los casos en los que $h(E) = +\infty$, claramente el estado analizado no está contenido en la solución, dado que no hay caminos abiertos pero existen nodos no visitados. Por ende, al no haber solución posible, la heuristica no sobreestima la solución.

Considerando este análisis, se puede afirmar que esta heurística nunca sobreestima el costo hasta la solución, por ende la heurística es admisible.

## 2.2. Heurística con poda direccional

El objetivo de esta y la siguiente heurística es eliminar estados que no pueden llevar a una solución, dado que con la aplicación previa de alguna regla válida se ha llegado a un estado

en el que algún casillero no visitado es inalcanzable. Por ende, dicho estado no puede formar parte de una solución.

La poda se realiza de la siguiente forma: dado el estado actual, para cada una de las direcciones válidas de movimiento (horizontal y vertical, centrándose en la última posición alcanzada), verificar si algún casillero libre es inalcanzable. Si esto se cumple, asignar $h(E) = +\infty$. Caso contrario, retornar el valor dictado por la heurística en 2.1.

De esta forma, se desestiman estados que no pueden llevar a una solución, efectivamente podando el espacio de búsqueda. Sin embargo, este proceso de poda considera que *es más probable* encontrar casilleros inalcanzables en las direcciones de movimiento dictadas por el problema, pero es posible que haya nodos inalcanzables en posiciones no analizadas por esta poda.

### 2.2.1. Admisibilidad

Esta heurística retorna el mismo valor que 2.1. si no se ha encontrado que algún casillero en las direcciones analizadas es inalcanzable. Si es inalcanzable, el estado analizado no forma parte de ninguna solución. Como la heurística en 2.1. es admisible, esta heurística lo es también.

## 2.3. Heurística con poda completa

En este caso, a partir del estado actual se analiza todo el tablero en búsqueda de casilleros inalcanzables, efectivamente eliminando todos los caminos que no llevan a una solución. Si no se encuentra un casillero inalcanzable, se retorna el mismo valor que en 2.1.

### 2.3.1. Admisibilidad

El análisis es idéntico al caso 2.2.

# 3. Búsquedas informadas

En esta sección se discutirán los resultados obtenidos con las búsquedas informadas (*Greedy* y *A**), utilizando las diferentes heurísticas mencionadas en la sección anterior.

Para comparar los resultados obtenidos del algoritmo *Greedy* con las diferentes heuristicas, se puede observar la tabla 2. Se analizaron casos con tableros de dimensión 9x9, con cantidad de formas y colores de entre 5 y 10 en cada caso.

Para empezar, cabe destacarse que no se ha podido encontrar diferencias en la cantidad de nodos expandidos, visitados o en frontera para la heurística con poda completa y la heurística con poda direccional. Esto puede querer decir que, en la práctica, son equivalentes. Las variaciones en el tiempo de cada caso pueden ser atribuídas a sutiles diferencias en la implementación, con la poda completa logrando mejores tiempos de ejecución en los casos con menor cantidad de nodos expandidos, y la poda direccional obteniendo mejor *performance* en los casos más *difíciles*.

Es precisamente en estos casos *difíciles* dónde se puede apreciar la ventaja que trae utilizar la poda. En la mayoría de los casos, las diferencias tanto en tiempo como en expansión de nodos para las heurísticas con o sin poda son pequeñas, o incluso inexistentes. Dónde se puede apreciar una gran diferencia es en estos casos dónde es necesario expandir una gran cantidad de nodos para encontrar una solución (resaltados en negrita en la tabla). En estos casos, se puede apreciar hasta un incremento de 10 veces en la cantidad de nodos expandidos. Por lo tanto, se puede afirmar que la poda es efectiva.

No se pudo encontrar una forma efectiva para producir estos casos *difíciles*, por lo que su hallazgo se realizó únicamente mediante una gran cantidad de muestras. Sin embargo, se encontró que son más frecuentes en tableros de dimensiones mayores. Es por eso que, si bien

los algoritmos con poda pueden resolver tableros de dimensiónes hasta $20X20$ con gran velocidad, se encontró que estos casos resultan imposibles de resolver en un tiempo razonable a dimensiones altas por estos algoritmos.

Otro detalle de sumo interés es la buena *performance* del algoritmo *greedy* en comparación a $A*$, obteniendo valores similares en los resultados. Esto se puede asociar a la calidad de las heurísticas, no solo en lo que respecta a la poda de estados que no llevan a la solución, sino a la correcta elección del valor de retorno. Claramente, los valores de retorno compartidos por las heurísticas analizadas dan una idea precisa de cuánto falta para llegar a la solución, valor intrínsecamente asociado con el $Remaining(E)$ discutido más arriba. Además de esto, se puede asumir que la buena *performance* confirma que el criterio tomado con respecto a los grados de los vértices da una buena noción de por dónde puede buscarse la solución. Al tender el costo heurístico a 0 a medida que un estado se aproxima a la solución, *Greedy* efectivamente lo seguirá expandiendo hasta el punto que consiga la solución, o bien que llegue a un estado no contenido en la misma.

Una optimización que cabe recalcar que ayudó a mantener el impacto en memoria mínimo para la búsqueda *greedy* y $A*$ es no introducir en la cola de prioridades − el conjunto de frontera − aquellos nodos cuyos estados en los que $h(E) = +\infty$. Esto es porque ese costo implica que la solución no se puede alcanzar siguiendo la rama de ese nodo, y por lo tanto mantenerlo en memoria no solo no tiene valor, sino que también es en detrimento del uso de memoria.

# 4. Conclusiones

Los resultados muestran que el peor rendimiento se obtuvo tanto con búsquedas BFS y DFS, mientras que los algoritmos A* y Greedy arrojaron resultados considerablemente mejores tanto en tiempo como en cantidad de nodos visitados, lo que significa que las heurísticas utilizadas fueron sumamente efectivas. Se decidió en las tablas no analizar los resultados del algoritmo IDDFS dado que, en el problema de *Chain Reaction* en particular, es, como se ha mencionado más arriba, equivalente a una búsqueda DFS con profundidad máxima establecida en la cantidad de casilleros no ocupados totales del tablero.

Cabe resaltar que 3 de los 4 algoritmos no consiguieron terminar con todos los casos de prueba. Tanto para búsqueda BFS como $A*$, la ejecución fue finalizada porque el sistema se quedó sin memoria durante la búsqueda (de un limite de 8 GB, fijado mediante el flag -Xmx8g de la JVM), mientras que, como se discutió en secciones anteriores, la ejecución en DFS fue finalizada forzosamente en los casos en que el tiempo de ejecución superaba el *timeout* de 10 minutos.

Que la explosión en el uso de memoria sea tan pronunciada para los algoritmos de búsqueda que progresan a lo ancho del espacio de soluciones (BFS y $A*$) y no para los que progresan en profundidad (DFS y Greedy) tiene sentido dada la naturaleza del problema: la máxima profundidad de la solución no solo está acotada, sino que es siempre exactamente la misma, es decir, la cantidad total de casilleros no vacíos. Esto quiere decir que una búsqueda en profundidad tomará nodos de la frontera mucho más frecuentemente que una búsqueda en anchura.