Gato y Ratón

Claudia Alvarez Martínez
Roger Moreno Gutiérrez
Jan Carlos Pérez González

Cuarto año, Ciencias de la Computación.
Facultad de Matemática y Computación, Universidad de La Habana, Cuba

1 Problema:

Un juego en un grafo no dirigido es jugado por dos jugadores, Ratón y Gato, que alternan turnos. El grafo se da de la siguiente manera: `graph[a]` es una lista de todos los nodos `b` tales que `ab` es una arista del grafo.
El ratón comienza en el nodo 1 y juega primero, el gato comienza en el nodo 2 y juega segundo, y hay un agujero en el nodo 0.
Durante el turno de cada jugador, deben viajar a lo largo de una arista del grafo que conecta con el nodo en el que se encuentran. Por ejemplo, si el ratón está en el nodo 1, debe viajar a cualquier nodo en `graph[1]`.
Además, no se permite que el gato viaje al agujero (nodo 0).
El juego puede terminar de tres maneras:

1. Si alguna vez el gato ocupa el mismo nodo que el ratón, el gato gana.
2. Si alguna vez el ratón llega al agujero, el ratón gana.
3. Si alguna vez se repite una posición (es decir, los jugadores están en la misma posición que en un turno anterior, y es el turno del mismo jugador para moverse), el juego termina en empate.

Dado un grafo, y suponiendo que ambos jugadores juegan de manera óptima, devuelve:

- 1 si el ratón gana el juego,
- 2 si el gato gana el juego, o
- 0 si el juego termina en empate.

2 Nuestra redefinición del Problema:

Sea $G = (V, E)$ un grafo no dirigido donde cada $v \in V$ está representado por un número entero. Definamos entonces al vértice v_i como el vértice al que se asigna el entero i. A partir de G se desarrolla un juego adversarial, que parte del siguiente estado inicial:

- El agujero está en v_0.
- Jugador 1 (Ratón) comienza en v_1.
- Jugador 2 (Gato) comienza en v_2.
- Jugador 1 hace el primer movimiento.

Restricciones:

- Los jugadores juegan por turnos.
- Un jugador que se encuentra en una posición v_i, puede moverse a la posición v_j, solo si $(v_i, v_j) \in E$.
- Un jugador en su turno debe moverse siempre.
- Jugador 2 no puede ocupar la posición v_0.

Definición de Estado: Definamos un estado del juego como una tripla $\langle v_i, v_j, f \rangle$, donde $v_i, v_j \in V$ y $f \in \{1, 2\}$. Los valores de v_i y v_j representan las posiciones que ocupan el jugador 1 y jugador 2 respectivamente; $f = 1$ si es el turno del jugador 1, y $f = 2$ si es el turno del jugador 2.

Estados finales del juego:

1. (**VICTORIA de Jugador 1**) Jugador 1 está en v_0.
2. (**VICTORIA de Jugador 2**) Ambos jugadores están en un mismo vértice v_k cualquiera.
3. (**EMPATE**) Ambos jugadores se encuentran en un **estado** en el que ya estuvieron antes.

Debido al carácter adversarial del juego estos estados finales son disjuntos, nunca se dará el caso en que ocurra más de uno al mismo tiempo.

3 Soluciones implementadas:

3.1 Un primer acercamiento: Minimax

Dado que el problema asume que ambos jugadores toman decisiones de manera óptima, una primera aproximación *naive* puede ser el uso del algoritmo *Minimax*.

Para ello debemos generar un grafo dirigido que represente todos los posibles **estados** del juego (es decir, el árbol de juego), mediante un recorrido en el que se comienza por el estado incial (el cual es único) y a partir de este se crean arcos hacia los posibles estados que genera el turno del jugador. Como resultado de este recorrido, se obtiene un grafo con todos los posibles **estados** del juego.

Es evidente que las hojas de este grafo corresponden a los estados terminales en los que el Jugador 1 o el Jugador 2 resultan victoriosos. Además, es importante destacar que en este grafo pueden existir ciclos cuando se repiten ciertos estados, es decir, cuando a partir de un estado se llega a otro que puede conducir de vuelta al estado original (no confundir con el estado inicial del juego). Según la definición del juego, estos ciclos constituyen un EMPATE. Vamos a convertir cada uno de esos ciclos en estados terminales, para ello, no pondremos el arco que cierra el ciclo, en cambio lo vamos a redirigir hacia un vértice hoja especial que va a representar el empate. Como resultado de cortar estos ciclos, el grafo resultante es un **DAG**, ya que es dirigido y no contiene ciclos.

Para obtener el valor de evaluación de una instancia del juego asumiendo que ambos jugadores juegan de forma óptima, lo que haremos es calcular el valor de la raíz de forma recursiva, aplicando la función de Minimax, partiendo partiendo de que las hojas pueden tener tres posibles valores (1 si gana el Jugador 1, 0 si hay empate, o -1 si gana el Jugador 2).

3.1.1 Análisis de Correctitud:

Demostremos la correctitud del algoritmo con inducción fuerte sobre la profundidad del árbol d, o sea, sobre la cantidad de turnos:

Caso base ($d = 1$):
Se tiene un árbol con una raíz y varias hojas, las cuales están resueltas por definición (1: Gana Jugador 1, 0: Empate, -1: Gana Jugador 2), entonces solo quedaría maximizar (porque es turno del Jugador 1) y retornar el valor.

Paso inductivo ($1 \leq k \leq d$): Supongamos que el algoritmo toma una decisión óptima para k, $1 \leq k \leq d$. Dado entonces un árbol de profundidad $d + 1$, se pueden dar dos posibles casos:

- Juega Jugador 1:
  Maximiza sabiendo que sus hijos (con profundidad a lo sumo d) se resuelven de manera óptima, por tanto la raíz del árbol de profundidad $d + 1$ se resuelve de forma óptima.
- Juega Jugador 2:
  Minimiza sabiendo que sus hijos (con profundidad a lo sumo d) se resuelven de manera óptima, por tanto la raíz del árbol de profundidad $d + 1$ se resuelve de forma óptima.

Entonces, queda demostrado que el valor de la raíz de cualquier árbol optimizado por el algoritmo de Minimax tiene el valor óptimo del juego, y por tanto la solución que se pide.

3.1.2 Análisis de Complejidad Temporal:

La complejidad temporal está dada en este caso por las dos operaciones importantes que se realizan: la creación del DAG y el proceso de optimización con *Minimax*:
El DAG construido en la solución tiene $2n^2$ vértices, y a lo sumo $\frac{n(n-1)}{2}$ arcos, y como su construcción se hace visitando todos los vértices, se resuelve en $O(n^2)$. La complejidad temporal de la función de optimización estaría dada por la cantidad de veces que juegan los jugadores, lo que sería la profundidad del árbol de juego, suponiendo que la profundidad del árbol sea d y que el cada estado (vértice del árbol) posee al menos dos hijos como promedio (branching factor) y sabiendo que el costo de cada nivel es constante porque lo que se hace es comparar resultados,

entonces la complejidad está dada por la siguiente fórmula:

$$1 + b + b^2 + ... + b^d = \frac{b^d + 1 - 1}{b - 1} = O(b^d) \tag{1}$$

lo que nos da un costo exponencial, ya que b está en función de n, por lo que sería $O(n^d)$.

3.2 Solución con Dinámica:

Una segunda aproximación, considerablemente más eficiente, puede lograrse mediante la aplicación de la *programación dinámica*, específicamente a través de la técnica de *memoization*. Este enfoque optimiza la solución al evitar la recalculación de subproblemas ya resueltos, lo que resulta en una reducción significativa del costo computacional.

El método parte de la misma estructura básica del algoritmo *naive*, pero cada vez que el algoritmo necesita resolver un estado, primero verifica si la solución para dicho estado ya ha sido almacenada en la estructura de *memoization*. Si es así, simplemente reutiliza el valor guardado; de lo contrario, el estado es procesado y el resultado es almacenado para futuras consultas. De esta manera, el algoritmo evita la redundancia en la computación de los subproblemas.

3.2.1 Análisis de Correctitud:

La incorporación de la técnica de *memoization* no altera la correctitud del algoritmo *naive*. Esto se debe a que *memoization* es simplemente una optimización que evita recalcular el valor de un estado previamente resuelto, almacenándolo en una estructura de datos para futuras consultas en tiempo constante.

En consecuencia, el resultado óptimo de un estado puede provenir de dos fuentes:

- Cálculo recursivo a partir de los estados hijos, tal como en la versión *naive*.
- Recuperación del valor almacenado mediante *memoization*, si dicho estado ya ha sido previamente calculado.

En ambos casos, el valor calculado para un estado es idéntico, ya que *memoization* no modifica la forma en que se determinan los valores de los estados, sino que simplemente optimiza el proceso de búsqueda. Por lo tanto, la incorporación de *memoization* no afecta la correctitud del algoritmo, únicamente mejora su eficiencia temporal.

3.2.2 Análisis de Complejidad Temporal:

Con el uso de *memoization*, garantizamos que cada estado del juego se evalúa como máximo una sola vez. Esto se logra al almacenar los resultados previamente calculados en una estructura de datos donde la inserción y consulta se realizan en tiempo $O(1)$.

Como el recorrido de la función de optimización se realiza sobre el DAG, sabemos que éste define un orden topológico de sus vértices, por lo que obtener el valor óptimo para un estado cualquiera e_i, depende excusivamente de estados e_j, tales que $i < j$ en el orden topológico.

Por la característica de evaluación en postorden de la función de optimización, cuando se analice un estado cualquiera e_i, podemos asegurar que todos los estados e_j de los que depende, ya han

sido optimizados. Esto permite que con la *memoization* se calculen y almacenen todos los valores una sola vez.

Dado que existen $2n^2$ estados posibles en el grafo del juego, y cada estado es procesado solo una vez (en $O(1)$ gracias a la consulta y almacenamiento mediante *memoization*), la complejidad temporal total del algoritmo es $O(2n^2) = O(n^2)$.

3.3 Solución con Dinámica y podas:

Para esta solución simplemente fue agregada una poda *alpha-beta*, clásica en algoritmos como *Minimax* para no continuar explorando ramas si ya se encontraba resuelto el padre, esto solo pasa para dos casos:

- Cuando se exploró una rama que retorna un 1 (máximo) y es el turno de maximizar del Jugador 1.
- Cuando se exploró una rama que retorna -1 (mínimo) y es el turno de minimizar del Jugador 2.

3.3.1 Análisis de Correctitud:

Como se analizó previamente, la poda *alpha-beta* simplemente elimina la necesidad de explorar ramas del árbol de decisión que no contribuyen al resultado óptimo. Esta técnica no altera el funcionamiento fundamental del algoritmo, manteniendo la lógica central intacta. La correctitud del algoritmo se preserva con esta modificación, dado que la poda se aplica únicamente a las ramas cuyo cálculo es innecesario para determinar el valor óptimo en el nodo raíz del subárbol correspondiente.

En otras palabras, si se eliminan ramas que no afectan la solución óptima en la raíz del subárbol, no se omite ninguna información relevante para el cálculo final. Dado que el criterio de poda se basa en evitar la evaluación de estados dominados (es decir, aquellos que no influyen en la decisión final), no se pierde precisión en la evaluación. Por lo tanto, la poda no introduce errores ni afecta el resultado final, lo que garantiza que el algoritmo siga siendo correcto tras esta optimización.

3.3.2 Análisis de Complejidad Temporal:

En la práctica, la poda puede hacer que el árbol se explore de manera más eficiente, porque evita realizar cálculos en ramas que no afectarán el resultado. En el mejor de los casos, la poda *alpha-beta* puede reducir el número de nodos evaluados de manera considerable, pero no dejará de ser O(n^2).