Estructura de Datos y Algoritmos

TIMBIRICHE

TRABAJO PRÁCTICO ESPECIAL

Maite Herrán (57609)

INTRODUCCIÓN

Timbiriche es un juego de tablero para dos o más jugadores el cual comienza con una grilla vacía de puntos sobre la que se turnan para agregar líneas horizontales o verticales entre dos puntos adyacentes no unidos. El jugador que completa el cuarto lado de un cuadrado de 1x1 gana un punto y toma otro turno. El juego continúa así hasta que no se puedan colocar más líneas, momento en el cual se define como ganador al jugador con más puntos. El tablero del timbiriche puede ser de cualquier tamaño.

El objetivo de este trabajo práctico es implementar dicho juego y crear una inteligencia artificial fundada en el algoritmo minimax que lo pueda jugar.

A continuación se detallaran las decisiones que fueron tomadas para la elección de las estructuras de datos utilizadas, para la implementación del minimax, y para la heurística de este algoritmo, entre otras cosas.

ALGUNAS ESTRUCTURAS

GENERAL DEL JUEGO

Se utilizó el modelo-vista-controlador (MVC) para desarrollar el software que permite jugar al timbiriche y así poder separar la lógica de la aplicación de la interfaz del usuario. Lo que hace esta arquitectura es dividir a la aplicación en tres partes:

- El modelo: maneja los comportamientos fundamentales y los datos de la aplicación. En este juego responde a solicitudes de información por parte del usuario como la cantidad de puntos que tiene cada jugador. Responde también a instrucciones para cambiar el estado de la información (siempre que sea posible) como cuando se quiere agregar una línea entres dos puntos adyacentes. En este caso el modelo es un conjunto de estructuras de datos y los métodos que las gestionan.
- La vista: es una representación visual del modelo que puede ser utilizada por un usuario. Actúa de filtro entre el modelo y el usuario para así, por ejemplo, dejar que este pueda ver el puntaje de los jugadores pero que no pueda modificarlos.
- El controlador: es el nexo entre el modelo y la vista. Recibe la entrada del usuario y realiza llamadas al modelo para luego poder responder a la vista.

TREE GENERATOR

Para la generación del dot se usó una estructura que se crea cada vez que el controlador llama a la inteligencia artificial cuando necesita que esta analize las jugadas posibles y devuelva la más conveniente. Como la consigna del trabajo dice que en la interfaz gráfica debe de haber “una opción para generar un archivo en formato dot con el árbol que fue explorado por el algoritmo [minimax] en la última jugada“, se decidió que, para cada jugada de la IA, se cree un string en el cual se vaya construyendo el árbol explorado y que solo se cree el archivo .dot con este árbol cuando el usuario apriete el botón “Generate Dot File”.

BOARD

Para representar al tablero, se utilizó una estructura que contiene una matriz de boxes (cajas). Cada box representa justamente las cajas que pueden ser formadas en el tablero luego de unir con líneas a vértices adyacentes. Estas líneas o aristas son representadas por campos booleanos en cada box. Es importante marcar que dos cajas contiguas comparten alguna de sus aristas, por lo que, cuando se agrega una arista en una caja, se debe tener en cuenta que también se agregó en la caja próxima que comparte a la misma. Se decidió utilizar una matriz de boxes por el hecho de que cada posición tiene acceso O(1) y también por la claridad hacia al programador. Se aprovechó la simplicidad que ofrece java en el diseño de clases.

STATE

Para poder llevar cuenta de las distintas etapas del juego y así poder realizar operaciones como deshacer la última jugada, se decidió tener una estructura llamada state en la cual se guardan todos los datos relevantes del juego en cada movimiento.

GAME

Cada state por el que haya atravesado será guardado en un stack de la estructura Game. Además se tendrá en un campo el state actual para así poder acceder a él y realizar cambios cuando lo requiera el usuario.

ALGORITMOS

Cálculo de states vecinos

Para poder obtener a las próximas jugadas posibles, se realizó una función que a partir de las líneas disponibles para dibujar que tiene un tablero, obtenga las jugadas posibles. Para esto llama a una función recursiva que lo que hace es recorrer todas las líneas posibles que se pueden agregar en el tablero, las va agregando de a una y va creando estados nuevos. Si una de esas líneas cierra un cuadrado, se llama a la misma función con el nuevo tablero con la línea agregada y se buscan los posibles próximos movimientos. Además, se va acumulando en una lista, las líneas agregadas para representar a la jugada completa. Al finalizar el turno del jugador, se crea un nodo que será utilizado en el algoritmo minimax con la próxima posible jugada y se agrega al set de estados o nodos vecinos. Digo estados o nodos porque en esta función lo que se hace es buscar a los estados vecinos, pero se aprovechó a crear a los nodos por el hecho de que serán utilizados en el algoritmo minimax. La estructura del nodo contiene básicamente al estado, al peso que será calculado en el algoritmo minimax y un campo que indica si el nodo fue podado o no.

MINIMAX Y PODA ALFA BETA

Para que la inteligencia artificial obtenga la jugada óptima a realizar, se utilizó el algoritmo minimax. El pseudocódigo de dicho algoritmo es el siguiente:

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node for the current player

    if maximizingPlayer
        bestValue := −∞
        for each child of node

            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
        return bestValue

    else    (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

Para mejorar el rendimiento del anterior algoritmo, se agregó al mismo la poda alfa beta. Este algoritmo busca disminuir la cantidad de nodos que son evaluados por el algoritmo minimax en su árbol de búsqueda. Deja de evaluar un movimiento cuando se ha encontrado al menos una posibilidad que demuestra que el movimiento es peor que un movimiento previamente examinado. El algoritmo mantiene dos valores, alfa y beta, que representan el puntaje mínimo que se asegura al jugador que maximiza y el puntaje máximo que asegura el jugador que minimiza, respectivamente. Inicialmente alfa es infinito negativo y beta infinito positivo, es decir, ambos jugadores comienzan con su peor puntaje posible. Siempre que la puntuación máxima garantizada del jugador minimizador (beta) sea inferior al puntaje mínimo garantizado por el jugador

maximizador (alfa) (es decir, alfa >= beta), el jugador que maximiza no necesita considerar a los descendientes de este nodo como nunca serán alcanzados en el juego real.

Porque la consigna pedía que en el árbol dot aparezcan los nodos que fueron podados, el algoritmo luego de encontrar que alpha es mayor a beta, no corta el for que estaba siendo recorrido, sino que lo sigue para agregar al buffer del dot los nodos podados. Sin embargo, lo sigue sin meterse en profundidad, sin llamar de vuelta al algoritmo recursivo.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        v := -∞
        for each child of node do

            if(not pruned)
                    v := max(v, alphabeta(child, depth – 1, α, β, FALSE))
                    α := max(α, v)
                    if β ≤ α then
                        pruned = true
        return v
    else
        v := +∞
        for each child of node do

            if(not pruned)
                    v := min(v, alphabeta(child, depth – 1, α, β, TRUE))
                    β := min(β, v)
                     if β ≤ α then
                          pruned = true
        return v
```

Para obtener la mejor jugada utilizando el minimax por tiempo, se utilizó una función timeSearch que llama a la misma función que es usada para obtener la mejor jugada por profundidad. Esta función lo que hace es ir llamando al minimax por profundidad cada vez con una profundidad mayor, y va guardando la mejor jugada. Cuando se acaba el tiempo, el minimax retorna null, indicandole a timeSearch que debe devolver como mejor jugada la calculada en la profundidad anterior por que en la profundidad actual, no alcanzó el tiempo para recorrerlo completo.

HEURÍSTICA

Se decidió como heurística una función que retorne la diferencia entre el puntaje del jugador que es el max en el minimax y el puntaje del jugador que es el min en el minimax. La profundidad a la que alcanza este algoritmo hace que esta heurística funcione. Sin embargo, si la profundidad en cuestión es 1, se agregó una mejora a la heurística, un cálculo que devuelve cuántas cajas con 3 líneas completadas tiene el tablero. Esto hace que el minimax no agregue una tercera línea cuando no llega a ver al próximo estado, que es uno en el cual el contrincante le cerrará la caja y ganará un punto.

CONCLUSIÓN Y OBSERVACIONES

Después de haber terminado el juego, se pudo ver que había una forma en la que se podía reducir el espacio de búsqueda y el tiempo cuando se trata de simetrías en el juego. Sin embargo, no se encontró una solución adecuada para este problema. Es por esto que el algoritmo minimax no evalúa las simetrías y es menos performante que algún código que lo haga. Como posibles extensiones del trabajo se podría, entonces, investigar más profundamente en formas de encontrar simetrías, y aplicar la solución en el algoritmo. También se podría hacer una implementación que acepte a más de 2 jugadores, a n jugadores.

Bibliografía

https://en.wikipedia.org/wiki/Minimax

https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning