Tiles Game

Un análisis sobre las estructuras y algoritmos utilizados

Emilio Tylson 54022 || Lucas Kania 54257 || Ignacio Gutierrez 54293

Indice

---

1. Introducción

Se propuso como Trabajo Práctico Especial la implementación del algoritmo Minimax para el juego "Azulejos". El objetivo del presente informe es exponer la manera en la que fue implementando el algoritmo, incluyendo las dificultades encontradas en el camino y sus respectivas soluciones.

Finalmente se realiza una comparación entre los tiempos que tarda en correr el algoritmo con y sin poda.

2. Estructuras utilizadas

La clase FileBoard es la encargada de leer un archivo de texto que representa un tablero y generar la matriz que luego representará el tablero.

El algoritmo minimax es ejecutado en la clase MiniMax quien tiene un método que, dado un Board y el puntaje actual de ambos jugadores, sabe devolver la mejor jugada acorde a los límites establecidos (limitación por tiempo/profundidad, con o sin poda, etc).

Para construir el árbol, la clase MiniMax utiliza subclases de una clase llamada Node. Node maneja internamente los datos necesarios para ejecutar el algoritmo. Las subclases previamente mencionadas son MinNode y MaxNode.

Para representar un tablero se utilizó la clase Board que internamente maneja una matriz de datos tipo char. Se optó por trabajar con chars (built-in) debido a que es el tipo de dato con menor peso (1 byte) que se puede manejar en java.

La clase State encapsula el estado actual de un juego. Internamente maneja una instancia de Board y otra de MiniMax, ademas de variables de tipo int para llevar registro de los puntajes y otra de tipo boolean para indicar de quien es el turno.

Tanto el display de consola (DisplayConsole) como el display gráfico (GDisplay) son clases que implementan la misma interfaz: Displayable.
Una clase que implementa Displayable, implementa un unico metodo: display(). Dicho método se encarga de mostrar, ya sea por consola o en la ventana, la información del juego.

Para regular los turnos de los jugadores se utilizó la clase Game. Esta clase maneja una instancia de las siguientes clases: State, GDisplay y MinimaxRunner. MinimaxRunner es una clase que implementa Runnable y tiene como único objetivo crear un thread que haga la llamada al método que ejecuta el algoritmo minimax.

Justificación de la clase MinimaxRunner: Ejecutar el algoritmo minimax desde el mismo thread que utiliza la interfaz gráfica entorpece el trabajo de la misma y como resultado la ventana se ve como si el juego se hubiese trabado hasta una vez finalizado el algoritmo.

Jerarquia de JPanels (Swing) :

- **GDisplay** hereda de JFrame y es la ventana del juego. Como estado interno tiene 2 clases que heredan de JPanel: BoardPanel y DataPanel.
- **BoardPanel**: Maneja una matriz de objetos tipo TilePanel. Se encarga de posicionar cada uno de ellos.
- **DataPanel**: Posiciona una instancia de la clase ScorePanel para mostrar el puntaje actual del juego.Ademáss es la clase encargada de mostrar el resultado final.
- **TilePanel**: Hereda de JPanel y hay una por cada tile en la matriz (incluyendo los espacios vacíos del tablero).

3. Algoritmo Minimax

Minimax es un algoritmo de toma de decisiones en un juego por turnos de dos jugadores, y se basa en minimizar la pérdida en el peor caso posible. Es un algoritmo por fuerza bruta, el cual recorre un árbol de todas las posibles jugadas con un recorrido DFS. El jugador que inicia el algoritmo, y es al que le interesa saber qué jugada tomar , se lo modela como Max, y al oponente como Min. El algoritmo sin limitaciones (por nivel ni tiempo) se recorre por profundidad hasta el caso base, en el cual no existan mas jugadas por hacer. Ahí se determina un valor heurístico que describe el suceso de jugadas hechas (se puede pensar que este número describe qué tan favorable es el conjunto de jugadas para Max, viendo a Max como referencia). A partir de ahí, Max debe elegir la jugada que prometa el mayor valor heurístico; y min debe elegir la jugada que determine el menor valor heurístico(es decir la jugada que favorezca menos a Max).

El algoritmo tiene el siguiente pseudocódigo:

```
función minimax(node, maximizingPlayer)
    if  node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, FALSE)
            bestValue := max(bestValue, val);
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, TRUE)
            bestValue := min(bestValue, val);
        return bestValue
```

En primer lugar se tomó la decisión de crear nodos para el árbol generado por el algoritmo, debido a que el trabajo exigía conservar el árbol para una posterior impresión. Como cada nodo tienen una estructura idéntica pero comportamiento diferente en situaciones como actualizar el valor heurístico , la forma de imprimirse en el formato .dot, e incluso en la inicialización de las valores .Se decidió modelar a Max y Min con nodos diferentes clases y con comportamiento propio ,cuya superclase era un clase abstracta que definía los métodos que cada Nodo debe tener y contiene el comportamiento común.

- **Clase abstracta Node:**

  - Posee todos los atributos de un nodo.
  - Determina el valor heurístico.

- **Clase MaxNode/MinNode:**

  - Sabe actualizar su value y su parámetro alpha y beta.
  - Sabe si tiene si tiene que podar o no.
  - Sabe en el valor heurístico agregar el puntaje al jugador determinado si. . corresponde a un nodo terminal.
  -Sabe generar nodos hijos. Si es Max genera nodos hijos Min, y viceversa.

En el trabajo se necesito hacer diferentes variantes del algoritmo Minimax, recorrido con y sin poda alpha-beta ,limitado por nivel, limitado por tiempo.

Debido a limitaciones de Heap y de tiempos de procesamiento, para juegos en que generar todo el árbol de decisiones de principio a fin es muy grande; se debe limitar al algoritmo Minimax por nivel o por tiempo. Y para una optimizar su recorrido en tiempo y espacio (es decir no generar nodos que se sabe que no son necesarios) una poda alpha-beta.

3.1 Recorrido limitado por nivel

La limitación por nivel es una condición en la que se corta (caso base de un recorrido recursivo) el recorrido DFS(condición agregada al caso base). Este nivel es controlado por una variable que representa el nivel , y se pasa como parámetro para indicar a qué nivel del árbol pertenece la actual llamada recursiva de la función minimax.

3.2 Poda alpha-beta

La poda alpha-beta permite optimizar el recorrido DFS del árbol sin ahondar en ramas que se saben que no van a influir en los nodos Min y Max. Cada nodo lleva variables que representan al parámetro alpha y al parámetro beta. El nodo Max se encarga de actualizar el parámetro alpha y el nodo Min el parámetro beta(comportamiento propio de cada nodo). Estos parámetros simbolizan la mejor jugada que se tomó previamente, es decir alpha es la mejor jugada de un ancestro Max y beta es la mejor jugada de un nodo ancestro Min tomada previamente.
Para entender el proceso de poda, se toma como referencia un nodo Max con fines explicativos:

Luego de recorrer a un hijo de forma DFS , actualizar el parámetro correspondiente del nodo actual(si es Max se actualiza alpha y se es Min se actualiza beta), y actualizar el valor de la mejor jugada hasta el momento del nodo. Si este valor (mejor jugada hasta el momento) es mayor que beta, no tiene sentido seguir recorriendo los demás hijos. Esto se debe que a lo sumo el próximo hijo va a mejorar la mejor jugada actual, pero el nodo padre Min (representado por beta) ya tiene una mejor jugada a elegir, que la que podría devolver el nodo actual(Max). Por lo tanto no tiene sentido recorrer a los demás hijos. Caso similar se produce en un nodo Min, con la diferencia que ocurre la poda si el valor de la mejor jugada es menor que alpha.

Por lo tanto la poda se implementó de forma tal que los parámetros alpha y beta se pasen como parámetro, cada nodo luego de la llamada recursiva, sepa actualizar su correspondiente parámetro, y que el nodo sepa si se debe cortar o no el recorrido.

Por otro lado como el algoritmo no se tiene que correr con esta poda siempre, se implementó un flag, que debe estar en “true” para que el nodo pueda cortar el recorrido.

3.3 Recorrido limitado por tiempo

El algoritmo minimax tiene un recorrido estrictamente DFS, y haciendo un único recorrido en un tiempo determinado, no se puede determinar cuándo cortar el recorrido por profundidad y analizar las demás ramas. Por lo tanto se optó hacer diversos recorridos limitados por nivel mientras de el tiempo especificado al minimax.

Un problema surgido fue ver que el tiempo podía acabarse a lo largo de un recorrido minimax a un determinado nivel. Por lo que se incluyó en el caso base ademas del corte por: nivel o si es un nodo terminal(no hay más jugadas por hacer), un corte por si el tiempo se acaba.

El problema ocurrido fue que el árbol resultante del minimax que se cortaba por el tiempo ,era obsoleto (estaba incompleto). En consecuencia se decidió guardar el árbol anterior y completo, por cada llamada nueva a un minimax . Y un flag (atributo de la clase) que se active en el caso base de dicho algoritmo, que indica que el minimax no pudo concluirse por falta de tiempo. Así puede determinarse si tiene que darse como respuesta la copia del árbol anterior

Resumiendo lo dicho en pseudocódigo, el algoritmo minimax sería:

```
function solution(node, lvl,alpha, beta, time)

   if node is a terminal node or depth = 0 or (timeLimit && time<0)
      endByTime=time<0? True:false
         return the heuristic value of node

      node.alpha=alpha
      node.beta=beta

      for each child of node
       node.update(minimax(child,lvl - 1,node.alpha,node.beta,time-timeCycle))
       if(pruneAlphaBeta && node.alphaBetaPrune())
          return node.bestValue
       update timeCycle

          return node.bestValue
```

Cabe destacar los flags timeLimit , pruneAlphaBeta, para determinar en qué forma se esta recorriendo el árbol y así saber si se debe o no cortar el proceso. Tambien se ve una variable “timeCycle” que permite calcular el tiempo entre cada llamada a la funcion solution.

El pseudocódigo para el recorrido por tiempo sería:

```
function solutionByTime(board, max_points, min points, time)
      lvl=0
     while !endByTime or hasTerminal value
        prev=initialNode
        initNode= newMaxNode(board,max_points, min_points)
        lvl++
       solution(initNode,lvl,initNode.alpha,initNode.beta,time-timeWhileCycle)
       update timeWhileCycle

     if endByTime
          initNode=prev
```

En el pseudocódigo se observa cómo se va llamando al algoritmo minimax(función solution) con distintos niveles. También se ve que el ciclo determinado por el while termina con “hasTerminalValue” , indica que si luego de recorrer el minimax la jugada encontrada es terminal(ya sea ganar o perder) el ciclo se corta antes de que termine el tiempo. Las razones de este procedimiento se verán en el apartado de Heurística.

Por último se ve una variable “timeWhileCycle” que calcula el tiempo entre ciclos del while, permitiendo tener el tiempo restante luego de cada llamada al algoritmo minimax.

3.4 Heurística

Al llegar a un nodo terminal(tablero donde no puede hacerse mas jugadas) , llegar un límite por nivel o por tiempo , se debe retornar un valor que determine que tan beneficioso (para Max) fue la jugadas hechas hasta el tablero actual. Cuanto menor es el valor, menos beneficioso es para Max; es decir Min toma el menor valor posible y Max el mayor valor posible.

En un principio, para determinar este valor, se tomó los puntos acumulados por el valor jugador Max menos los puntos acumulados por el jugador Min. Pero mediante este algoritmo no podía determinarse si la mejor jugada posible elegida era una en la que Max perdía o ganaba . Y en consecuencia, en la limitación por tiempo no podía determinarse si era necesario realizar el algoritmo minimax con el próximo nivel de profundidad. Ya que seria innecesario si en anterior recorrido dieron todos los nodos terminales(se ganó o perdió en todos los nodos hojas). Para graficarlo con un ejemplo: si en el en el recorrido del minimax con limitación a nivel 3 el árbol resultante todos sus nodos eran terminales, en especial en todas las hojas pierde; mientras haya tiempo se seguirá recorriendo el algoritmo con nivel 4 , 5, .. así sucesivamente. Pero esto no era necesario, porque el árbol no puede crecer mas de tres niveles.

Entonces se necesitó determinar con un valor heurístico especial si se gano o perdió (refiriendo al Max). Primero se optó por tomar +inf como ganar y – inf como perder. En consecuencia el recorrido limitado por tiempo se cortaba si luego de correr el minimax devuelve -inf(la mejor jugada posible conlleva a perder, indicando así que todos los nodos terminales Max pierde) o + inf (si encuentra una forma de ganar).

Luego de diversas pruebas con esta heurística se noto en que si todas las jugadas le tocaba perder a la computadora(la que realiza el algoritmo Minimax), ésta optaba por realizar un jugada al azar. Pero esto conllevaba que la jugada que hacía no era la más efectiva posible.

Para solucionar este problema se decidió adoptar una marca que distinga si se ganó o perdió (para saber si cortar el recorrido mínima limitado por tiempo antes) y que a su vez haya distinta variaciones de esta marca de forma tal, que si en todas las posibilidades le toca perder elija la más favorable (pensando en referencia a Max). Se implementó que si se gana, el valor heurístico sea +inf/2 +p (p>0 porque los puntos de Max son mayores a los de Min) y si se pierde -inf/2+p (p<0). Por lo tanto mientras más negativo el valor -inf/2 , Max pierde con mayor diferencia. Y mientras más positivo +inf/2 Max gana con mayor diferencia.

De esta forma en la limitación por nivel, se puede cortar si luego de recorrer el algoritmo minimax se retorno como valor heurístico un número menor a -inf/2 (todas las posibles jugadas de Max implican que va a perder) o un número mayo a +inf/2 (Max encontró una jugada en la que va a ganar).

4. Análisis en tiempo del algoritmo minimax

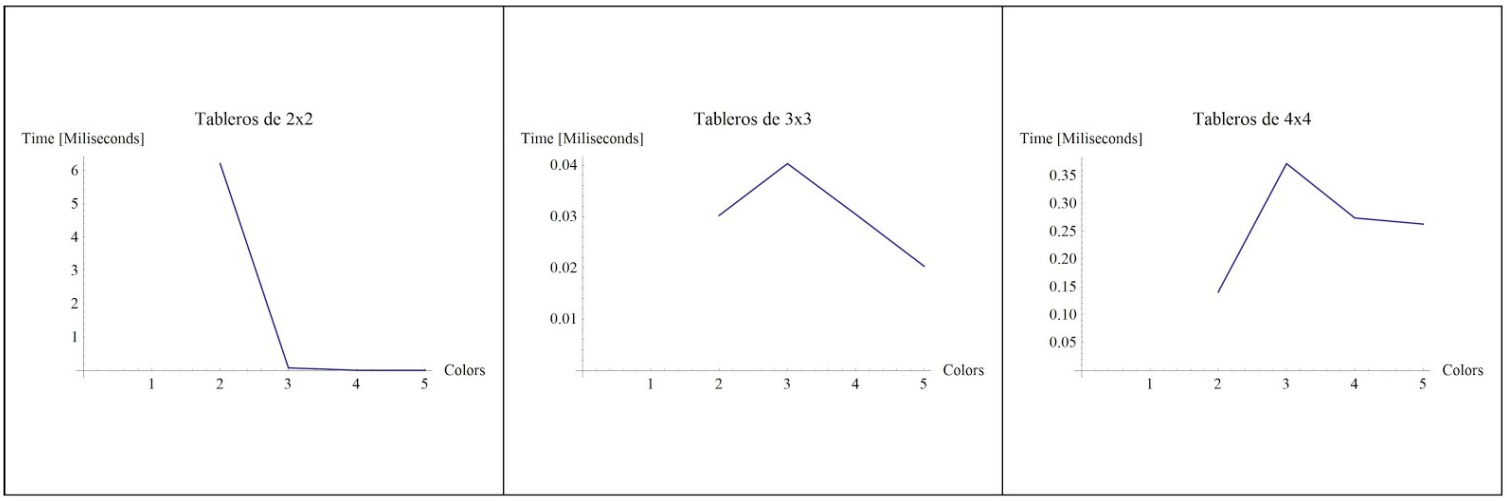
Debido a que el algoritmo **`solutionByTime`** de la clase **`MiniMax`** esta basado en utilizar el método solution, lo que analizamos acá es como se comporta el algoritmo minimax sin límites de tiempo, variando 3 factores, el tamaño del tablero, la cantidad de colores en el mismo y la profundidad máxima que puede alcanzar el game tree.

Todas las pruebas se basan en los resultados de la clase **`TestLevelPruneHeuristic`** que genera un archivo .txt con separación de campos por espacio, el cual se puede cargar en cualquier procesador de hoja de cálculo si se lo desea observar.

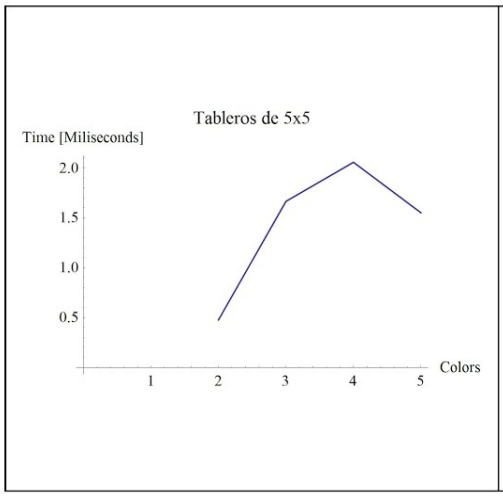
4.1 ¿Cómo impacta la cantidad de colores del tablero en el rendimiento del minimax?

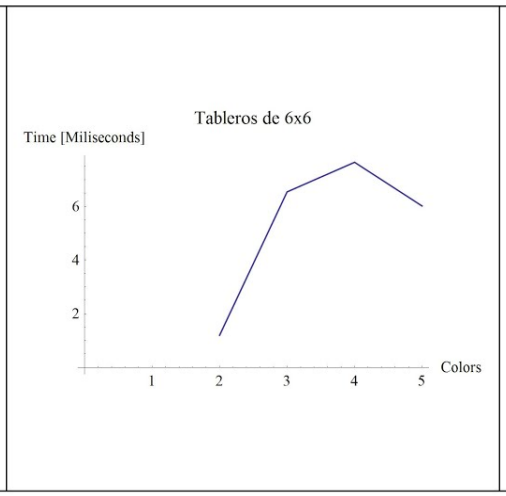
Para este análisis la la profundidad del game tree máxima se fijó en 4 niveles, de esta manera las únicas variables son el tamaño del tablero y la cantidad de colores.

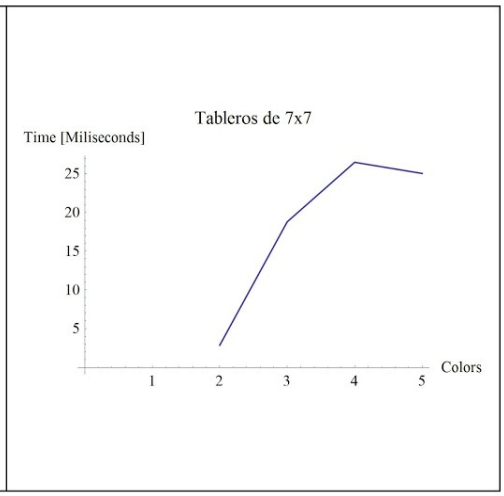
Tener en cuenta que los resultados en tiempo que se ven, no son de una única prueba con cada tipo de tablero, sino que son 100 pruebas con cada uno, siendo valores observados el valor promedio de la muestra.

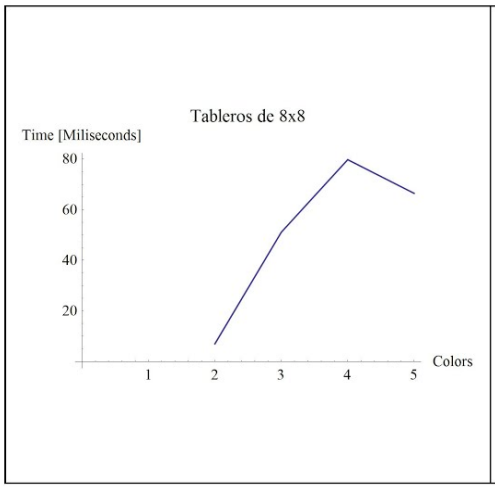


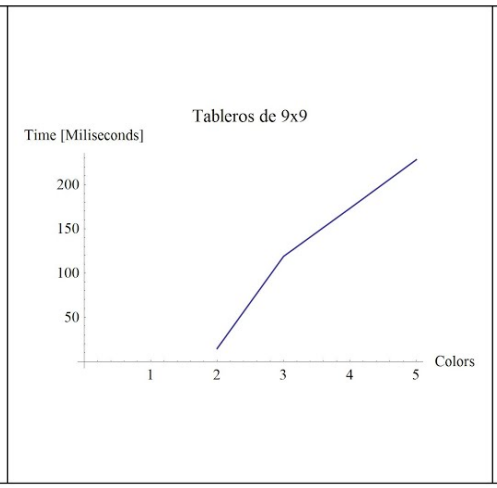
Podemos observar que para tableros pequeños como 2x2, 3x3 y 4x4, a partir de un cantidad de colores decrece el tiempo que conlleva encontrar una solución, esto es lógico pues tener más de tres colores en tableros tan pequeños genera que haya poca jugadas posibles y por lo tanto implica que el game tree es mucho más pequeño y por lo tanto se encuentra una solución en un menor tiempo que con pocos colores.

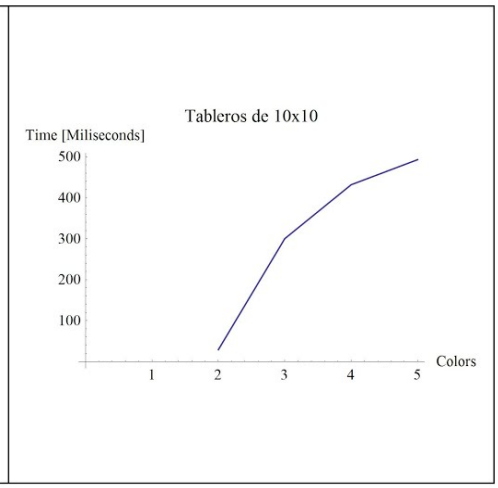


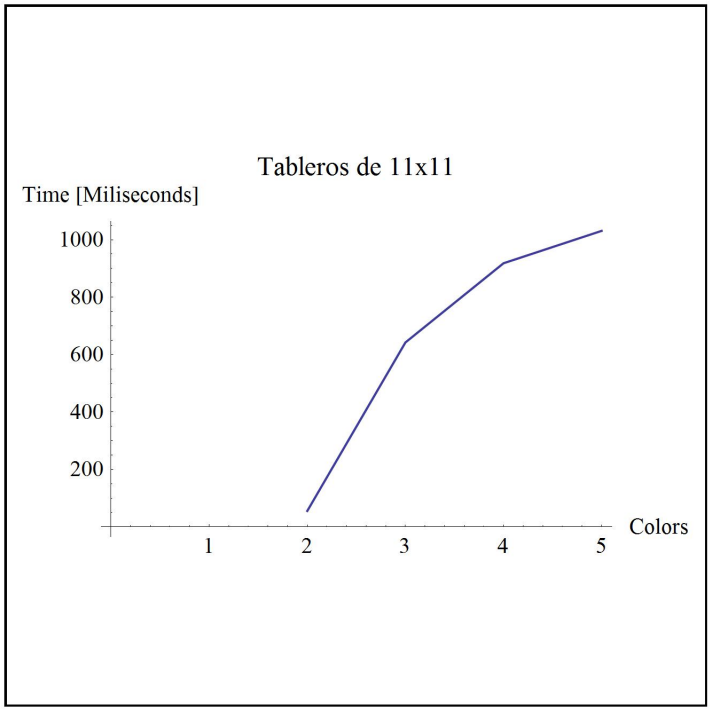










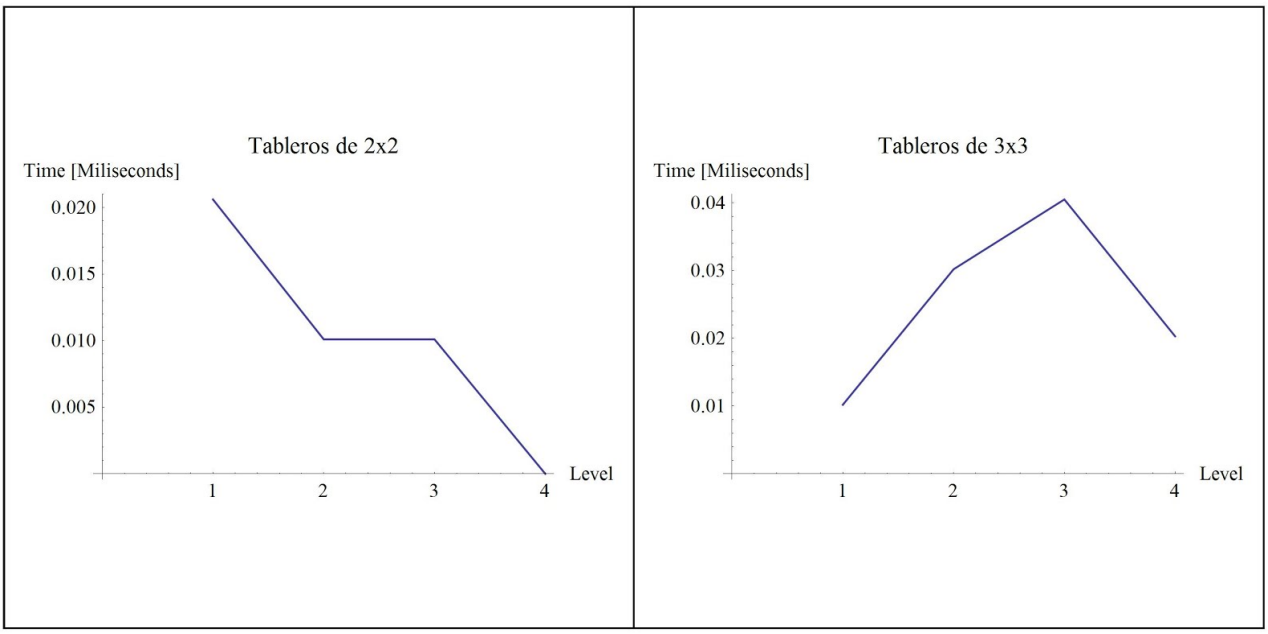


Podemos ver que para matrices de mayor tamaño sigue existiendo una cantidad de colores a partir de la cual sucede lo anteriormente descrito, pero se muestra claramente que el tiempo tiene un crecimiento logarítmico, lo cual puede verse perfectamente en tableros de 10x10 y 11x11.

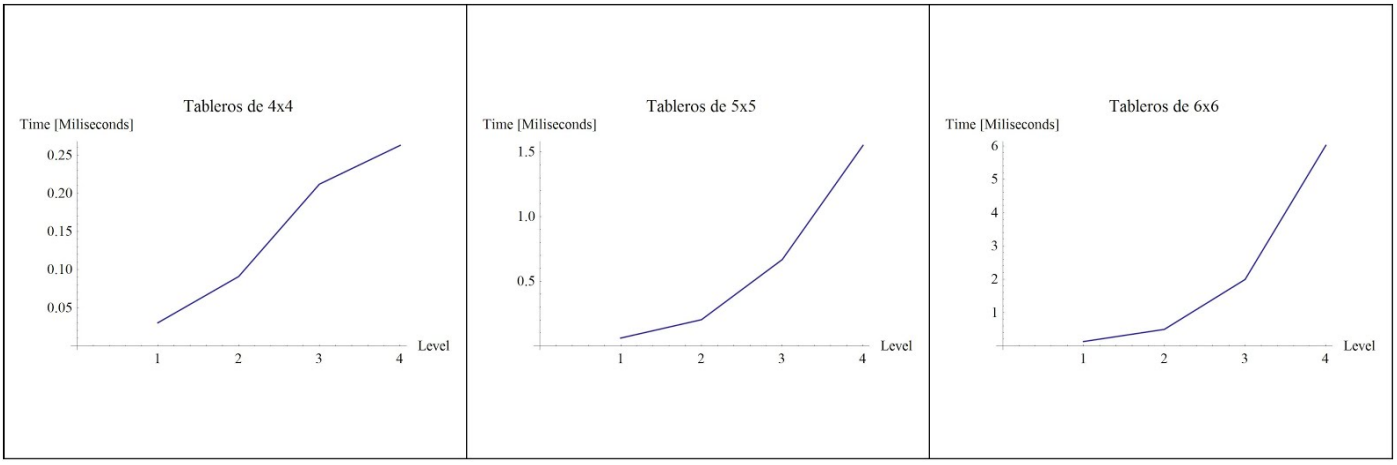
4.2 ¿Cómo impacta la profundidad del game tree en el rendimiento del minimax?

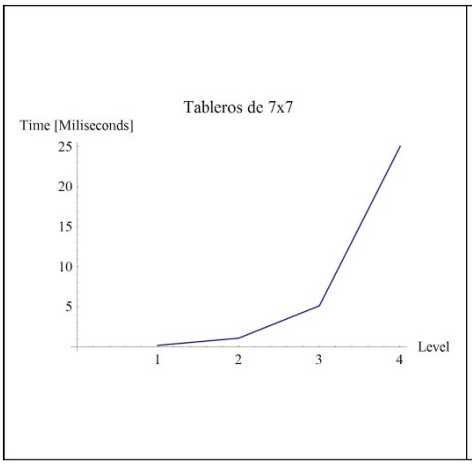
Para este análisis se fijó la cantidad de colores en 5, de esta manera las únicas variables son el tamaño del tablero y la profundidad del game tree.

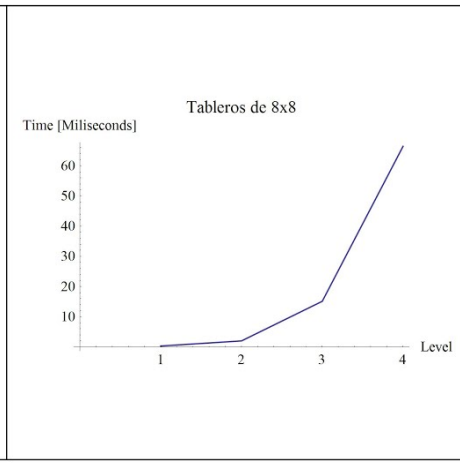
Observamos que el tiempo para analizar cualquier tipo de tablero con un solo nivel de profundidad es despreciable, pues es una análisis de qué jugada es la mejor sin ver “a futuro”.

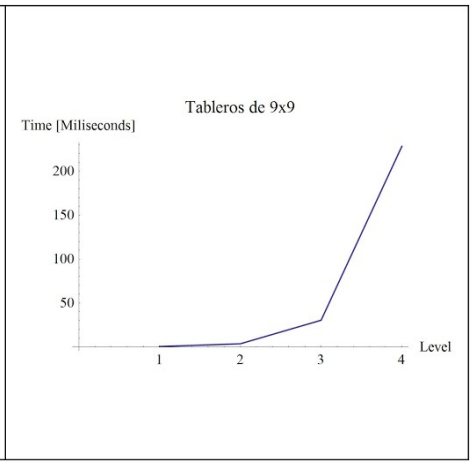


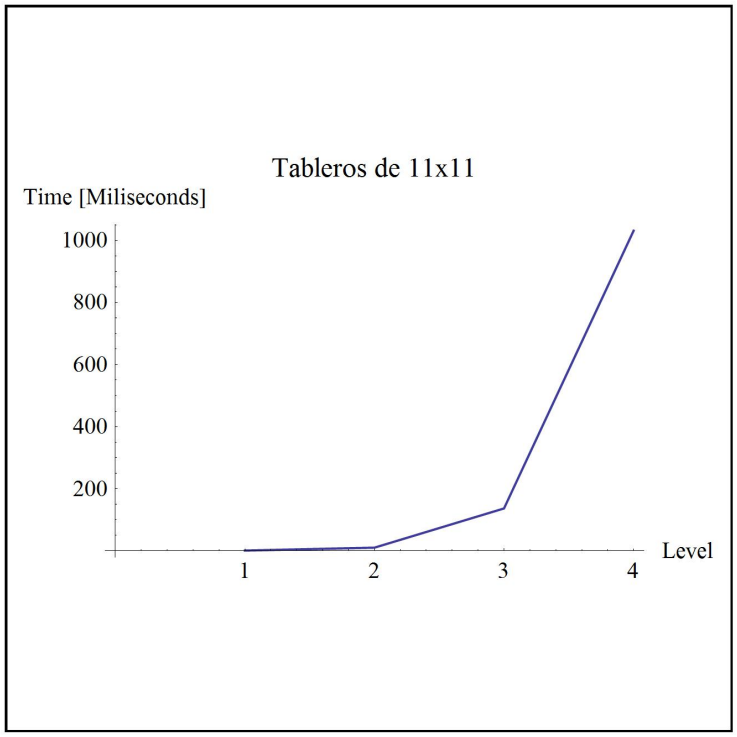
Para tableros de 2x2 y 3x3 podemos ver que se llegue a un tiempo constante, esto es debido a que el game tree se terminó de recorrer completamente en un nivel de profundidad anterior, lo cual tiene sentido para un tablero de 2x2 o 3x3 que tiene pocas posibilidades.











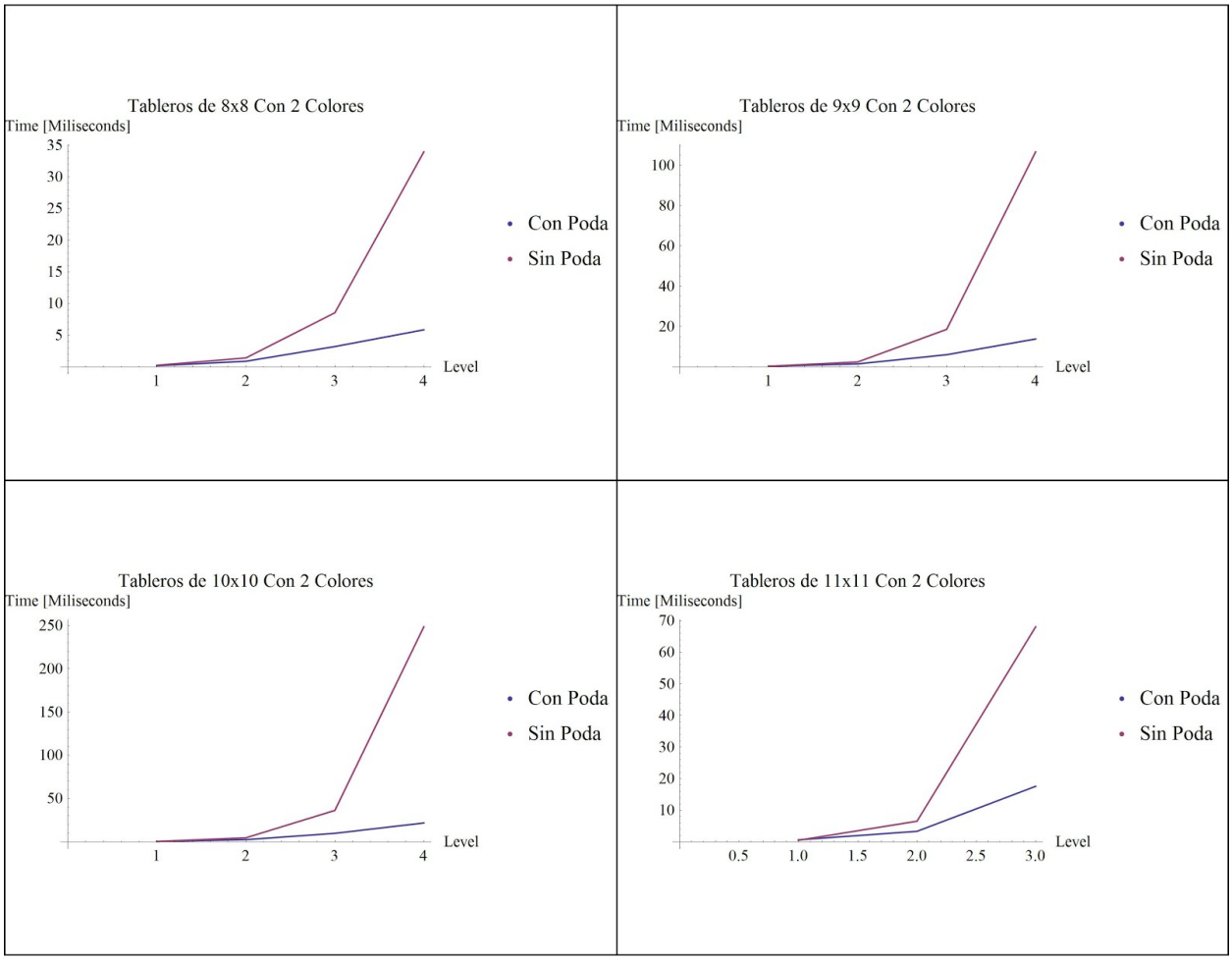
Podemos ver que para el resto de los tableros el crecimiento del tiempo es exponencial, y que rápidamente empieza a crecer a partir del nivel 2 de profundidad, indicándonos que cada nivel llevará muchos tiempo en procesarse que los anteriores.

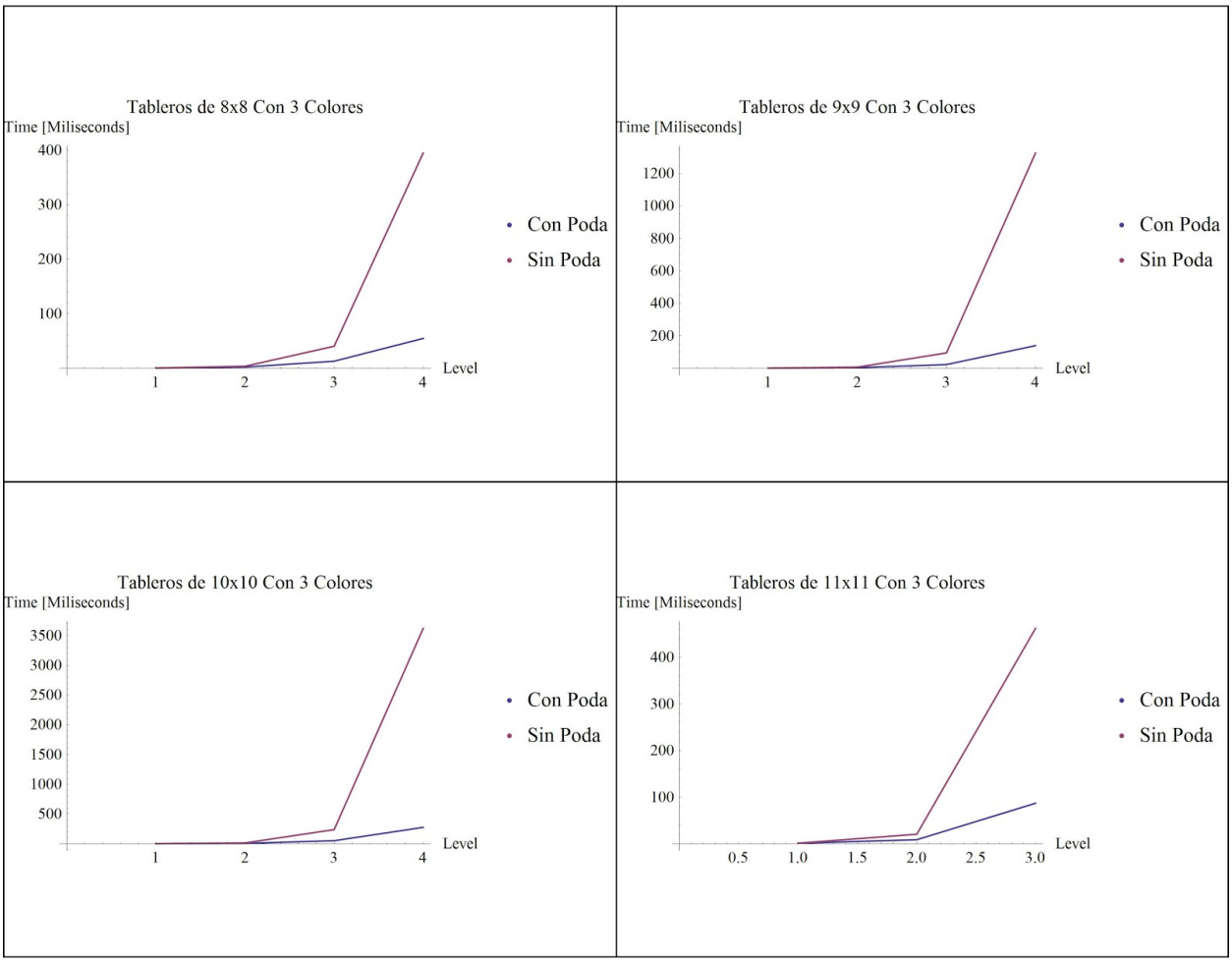
4.3 Con Poda contra Sin Poda

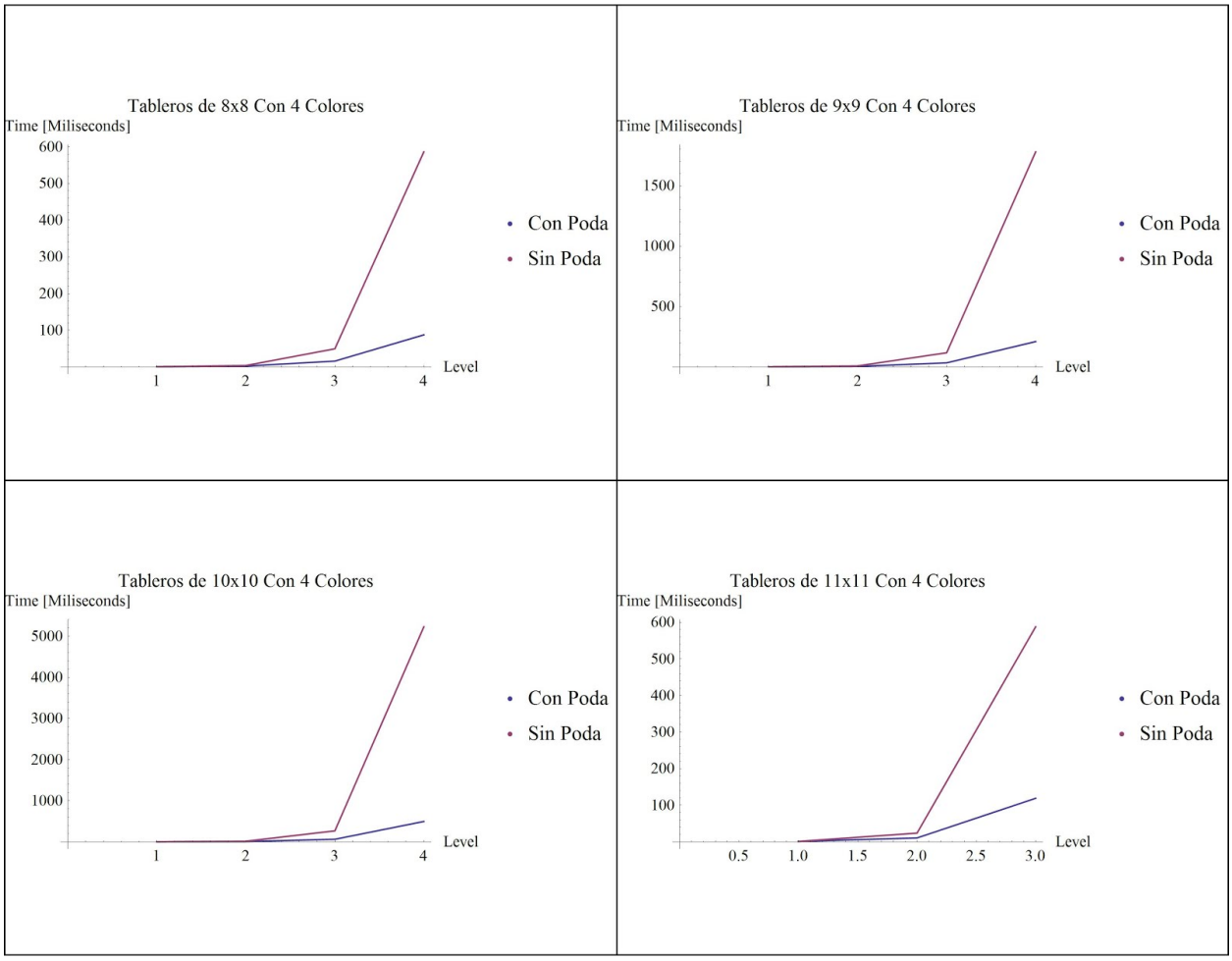
Todas las pruebas se basan en los resultados de la clase **`TestPruneVSNoPrune`** que genera un archivo .txt con separación de campos por espacio, el cual se puede cargar en cualquier procesador de hoja de cálculo si se lo desea observar.

Los tableros analizados fueron los más grandes, de esta manera se puede apreciar mejor la diferencia entre el algoritmo con poda y sin poda. Pero se puede verificar en el dataset usado que los resultados fueron siempre los mismos para cualquier tipo de tablero.

Tener en cuenta que los resultados en tiempo que se ven, no son de una única prueba con cada tipo de tablero, sino que son 100 pruebas con cada uno, siendo valores observados el valor promedio de la muestra.




Tableros de 8x8 Con 3 Colores
Time [Miliseconds]
400
300
200
100
1
2
3
4
Level
Con Poda
Sin Poda
Tableros de 9x9 Con 3 Colores
Time [Miliseconds]
1200
1000
800
600
400
200
1
2
3
4
Level
Con Poda
Sin Poda
Tableros de 10x10 Con 3 Colores
Time [Miliseconds]
3500
3000
2500
2000
1500
1000
500
1
2
3
4
Level
Con Poda
Sin Poda
Tableros de 11x11 Con 3 Colores
Time [Miliseconds]
400
300
200
100
0.5
1.0
1.5
2.0
2.5
3.0
Level
Con Poda
Sin Poda



Los resultados obtenidos fueron los esperados, las diferencias en tiempo entre con poda y sin poda son notorias en todo tipo de tableros.

Es importante resaltar dos resultados, cuando la profundidad máxima del game tree es de 3 o 4 niveles, las diferencias entre sin poda y con poda son muy grandes, mostrando la importancia del método para poder responder en un tiempo razonable.

Por último, cuando la profundidad máxima del game tree es baja (1 o 2), el tiempo que tarda el algoritmo con y sin poda es casi el mismo,en particular, con un único nivel de profundidad podemos ver que el algoritmo sin poda llevó menos tiempo que el mismo con poda, esto es debido a que al algoritmo sin poda se ahorra todas las comparaciones que para un único nivel de profundidad no tienen utilidad y por lo tanto termina antes.

4.4 Conclusiones Parciales

A partir del análisis anterior, podemos ver que claramente el impacto de la profundidad del árbol es mucho mayor que el de la cantidad de colores, pues la tendencia del tiempo a medida que aumentamos la profundidad del game tree es exponencial.

No se debe perder de vista que un tablero grande con muchos colores, generara árboles muy anchos, y por lo tanto cada nivel del game tree tardará mucho más en generarse y recorrerse. Las tres variables se potencian entre sí (Tamaño del tablero,Cantidad de colores,Profundidad Máxima).

Notas:

- Se utilizó Wolfram Mathematica 9.0 para el análisis de la información y generación de gráficos. (Los archivos utilizados se encuentran en la carpeta resources)
- Los datasets específicos utilizados en esta sección se encuentran en la carpeta resources.
- Las diferentes tablas de está sección se encuentran también en la carpeta resources.
- Los tableros utilizados para las pruebas no se encuentran almacenados en resources, pues se utilizaron tableros aleatorios, se puede ver el código en las clases del package massive_test mencionadas (Básicamente se le pide a board que genere un board aleatorio).

5. Conclusiones

La jerarquía de clases diseñada facilitó la implementación del algoritmo en diversos aspectos. Entre otros cabe destacar la relación de herencia entre la clase Node y sus hijos, que hizo posible manejar a todos los nodos con una misma colección.

El minimax es un algoritmo que se puede desarrollar sin previamente pensar en la heurística que se utilizara, pero es ahí donde reside una de las grandes dificultades, que determina que un estado de juego sea mejor que otro. La única forma de obtener una buena heurística es a partir de sucesivas pruebas.

En si la implementación del minimax expuesta no es la estándar, pues aquí se desarrolla un árbol de juego verdadero (para su posterior impresión), a diferencia del método tradicional donde se recorren los posibles estados de juego pero no se mantiene los mismos sino solamente el mejor estado hasta el momento.

En análisis en tiempo del minimax mostró que cuando la profundidad del árbol de juego creado es muy baja, el algoritmo tarda más con poda que sin poda, esto se debe a las múltiples comparaciones de alpha y beta que realiza con poda. Pero cuando la profundidad es mucho mayor, las diferencias en tiempo empiezan a aumentar.

Cabe remarcar que según los análisis previos pareciera ser que la variable que más pesa en el tiempo que tarda el algoritmo (existen tres variable: tamaño del tablero, cantidad de colores y profundidad del árbol), es la profundidad máxima a la que puede llegar el árbol.