1. Objetivo:

- Aplicar la teoría de árboles para el diseño de un agente capaz de jugar el tic-tac-toe (juego del gato o tres en raya) con un bajo costo computacional y un desempeño mejor que el promedio de las personas.

2. MARCO TEÓRICO – Algunos conceptos

Al investigar sobre las mejores jugadas para ganar en tic tac toe podemos crear un árbol con muchas ramificaciones, y en cada ramificación existirán nodos con todas las jugadas posibles.

La idea principal para lograr que el agente gane la partida es que se anticipe a la jugada de su oponente y use una de las jugadas indicadas en los nodos del árbol.

Para generar el árbol del proyecto usamos 'estructura de datos', listas enlazadas y teoría sobre los 'árboles'.

2.1 ÁRBOL DE DECISIÓN

Los árboles de decisión son herramientas excelentes para ayudar a realizar elecciones adecuadas entre muchas posibilidades. Su estructura permite seleccionar una y otra vez diferentes opciones para explorar las diferentes alternativas posibles de decisión. En este caso los posibles resultados corresponden a diferentes alternativas de uso potencial de la tierra.

Los árboles de decisión son guías jerárquicas multi-vía donde los valores de las características son el criterio diagnóstico para evaluar la calidad de la tierra y determinar el uso más apropiado de la tierra.

La jerarquía se refiere a que la toma de una decisión o camino lleva a otra, hasta que todos los factores o características involucradas se hayan tomado en cuenta. Es multi-vía porque pueden existir más de dos opciones y es una guía porque al responder una pregunta se llega a una decisión.

2.2 ÁRBOL DE JUEGOS

El árbol de juego es una representación de un juego que describe la estructura temporal de un juego en forma extensiva. El primer movimiento del juego se identifica con un nodo distintivo que se llama la raíz del juego. Una jugada consiste en una cadena conectada de ramas que comienza en la raíz del árbol y termina, si el juego es finito, en el nodo terminal. Los nodos representan los posibles movimientos en el juego. Las ramas que parten de los nodos representan las elecciones o acciones disponibles en cada movimiento. A cada nodo distinto del nodo terminal se le asigna el nombre de un jugador de modo que se sabe quién hace la elección en cada movimiento. Cada nodo terminal informa sobre las consecuencias para cada jugador si el juego termina en ese nodo.

El análisis por árbol de decisiones es una técnica que consiste en desmenuzar todos los caminos y alternativas posibles hasta llegar a los distintos esta dos finales. El esquema se

representa en forma de árbol, en el que a cada rama se le asigna una probabilidad de ocurrir.

2.3 NODO

Nodo: cada uno de los elementos de un árbol binario se denomina nodo. Un árbol Binario puede tener cero nodos y este caso se dice que está vacío. Puede tener un sólo nodo, y en este caso solamente existe la raíz del árbol o puede tener un número finito de nodos. Cada nodo puede estar ramificado por la izquierda o por la derecha o puede no tener ninguna ramificación.

Con relación al tipo de nodos que hacen parte de los árboles, se identifican algunos nodos:

- *Nodo hijo:* cualquiera de los nodos apuntados por uno de los nodos del árbol.
- *Nodo padre*: nodo que contiene un puntero al nodo actual.

En cuanto a la posición dentro del árbol se tiene:

- *Nodo raíz:* nodo que no tiene padre.
- *Nodo hoja:* nodo que no tiene hijos.

En relación a su tamaño:

- *Orden:* es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, se dice que un árbol en el que cada nodo puede apuntar a otros dos es de orden dos, si puede apuntar a tres será de orden tres y así sucesivamente.
- *Grado:* el número de hijos que tiene el elemento con más hijos dentro del árbol.
- *Nivel:* se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz siempre será cero y el de sus hijos uno. Así sucesivamente.
- Altura: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también se puede hablar de altura de ramas.

2.4 LISTA ENLAZADA

Es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias (punteros) al nodo anterior y/o posterior.
El principal beneficio de las listas enlazadas respecto a los array convencionales es que el orden de los elementos enlazados puede ser diferente al orden de almacenamiento en la memoria o el disco, permitiendo que el orden de recorrido de la lista sea diferente al de almacenamiento.
Una lista enlazada es un tipo de dato auto-referenciado porque contienen un puntero o link a otro dato del mismo tipo. Las listas enlazadas permiten inserciones y eliminación de

nodos en cualquier punto de la lista en tiempo constante (suponiendo que dicho punto está previamente identificado o localizado), pero no permiten un acceso aleatorio.

2.5 BÚSQUEDA MINIMAX

Método básico propuesto para recorrer un árbol de juegos y seleccionar un movimiento más prometedor.
En el nivel de maximización se busca un movimiento que lleve a un número positivo grande. En el de minimización uno que lleve hacia negativos.

- Las Decisiones del maximizador deben tener conocimiento de las alternativas disponibles para el minimizador del nivel inferior, y al revés.
- Cuando se alcanza el límite de exploración, el evaluador estático proporciona una base directa para la selección de alternativas.
- Minimax Propaga información de abajo hacia arriba.

Para efectuar una búsqueda mediante minimax:

- Si el límite de búsqueda se ha alcanzado, calcular el valor estático de la posición actual. Dar a conocer el resultado.
- De otro modo, si el nivel es de minimización, usar minimax en los hijos de la posición actual y dar a conocer el menor de los resultados.
- De lo contrario, si el nivel es de maximización, usar minimax en los hijos de la posición actual y notificar el mayor de los resultados.

Si La profundidad máxima del árbol es m, y hay b movimientos legales en cada punto, entonces:

- La complejidad en tiempo del algoritmo minimax es O(b^m)
- La complejidad en espacio es O(bm) Para un algoritmo que genere todos los sucesores a la vez ó O(m) Para un algoritmo que genere los sucesores uno por uno.

Esta estrategia no fue la usada en el proyecto, por lo que no se profundizará más en la misma. Sin embargo queda constancia, para alguien que lea este informe y desee replicar este proyecto llamado 'Tic Tac Toe', véase más información de esta estrategia en la bibliografía (el libro de matemáticas discretas de Richard Johnsonbaugh).

3. Trabajo en equipo en un grupo de programadores

Al momento de trabajar en un proyecto de nivel considerable que requiere la colaboración de todos los programadores, surgió una necesidad, la de llegar a una forma de 'convivencia' entre nuestros códigos. Sin embargo, **como aporte**, hacemos un pequeño apartado en el que damos una recomendación a los futuros novatos que deseen aprender más sobre trabajo en equipo al momento de diseñar un programa que constantemente va cambiando.

3.1 Controlador de versiones

Un problema al momento de modificar un programa es que los demás compañeros del grupo pueden 'perderse' y el constante cambio de versiones "1.1, 1.2, 1.2.1…" causa confusión y

desorganización. Es por ello que trabajar con un controlador de versiones (como GIT) permite un trabajo mucho más organizado y tenemos además la ventaja de 'volver en el tiempo' entre las versiones, si algo salió mal en alguna versión '2' podemos regresar a la versión '1' e indagar que ha sucedido.

En general un controlador de versiones es un sistema que permite hacer cambios a través del tiempo a un producto, manteniendo las características obsoletas y al mismo tiempo guardando los cambios que se realizan como sub-versiones. El uso de esta herramienta ayuda de gran manera a un grupo de trabajo, al llevar un registro histórico de todo lo que se ha realizado en un proyecto y los cambios aplicados a través del tiempo, de manera que todo se guarda como un historial con todos los cambios que se han realizado, así el trabajo en equipo se vuelve más manejable.

3.2 Git

Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es decir Git nos proporciona las herramientas para desarrollar un trabajo en equipo de manera inteligente y rápida y por trabajo nos referimos a algún software o página que implique código el cual necesitemos hacerlo con un grupo de personas.

Algunas de las características más importantes de Git son:

- Rapidez en la gestión de ramas, debido a que Git nos dice que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente.
- Gestión distribuida; Los cambios se importan como ramas adicionales y pueden ser fusionados de la misma manera como se hace en la rama local.
- Gestión eficiente de proyectos grandes.
- Re-almacenamiento periódico en paquetes.

Es importante mencionar que las normas generales aprendidas sobre los lenguajes de programación (escribir una clase con la primera letra mayúscula, darle un nombre significativo a las variables: no como 'aux1…', sino algo representativo, de tal forma que quede autodocumentado). El uso de clases descriptivas (abstractas e interfaces en Java) como forma obligatoria de organización también es buena idea, tomar esto en cuenta si se usan otros lenguajes además de Matlab.

4. Implementaciones en el agente, por parte de los estudiantes

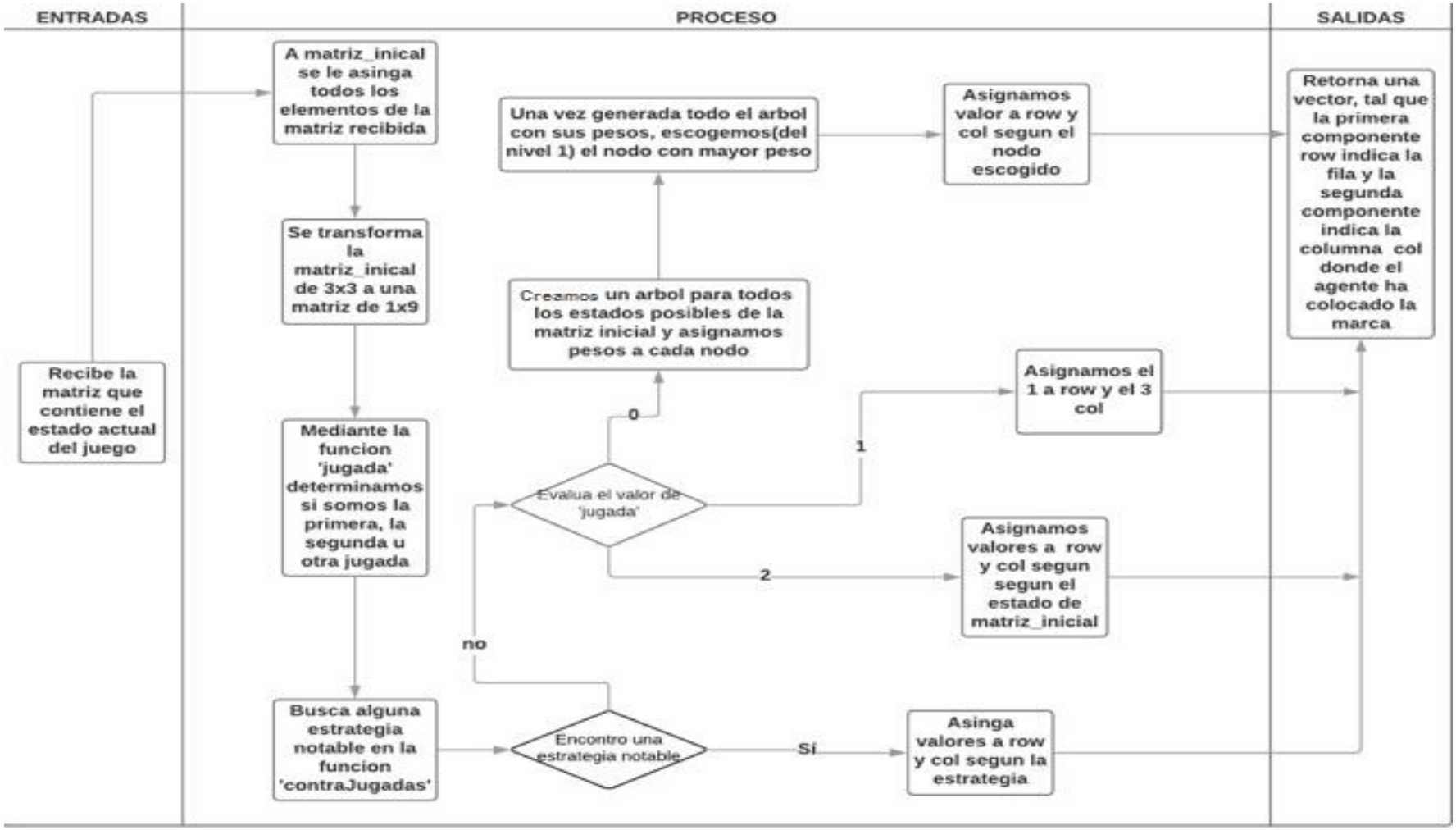
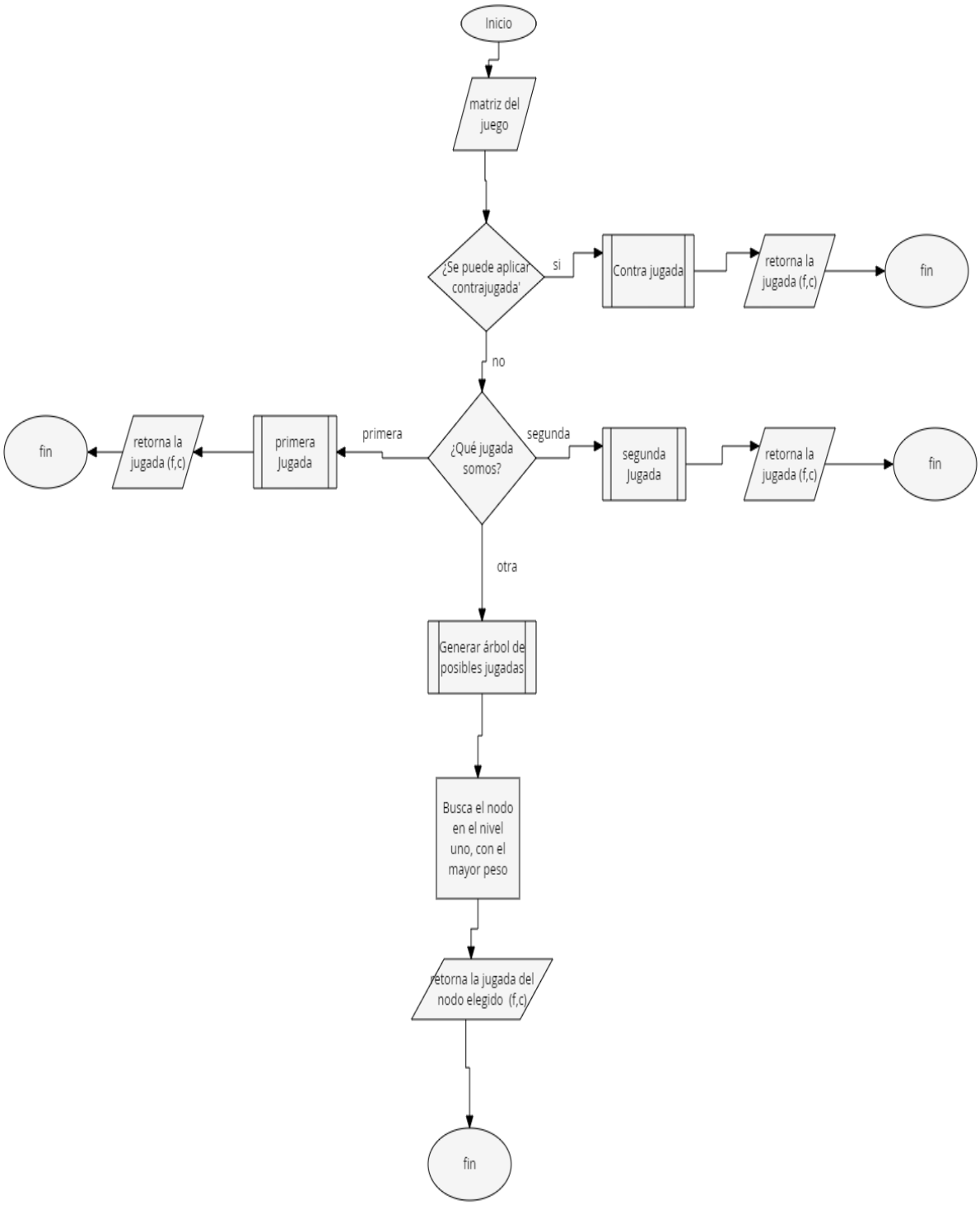


Diagrama 1, diagrama de flujo de *playAgentStudent.m*

Escuela Politécnica Nacional
Proyecto Tic Tac Toe

Otra forma de ver el mismo diagrama de flujo de *playAgentStudent*



Función que genera todas las posibles jugadas del 'tres en raya' a partir de una matriz dada

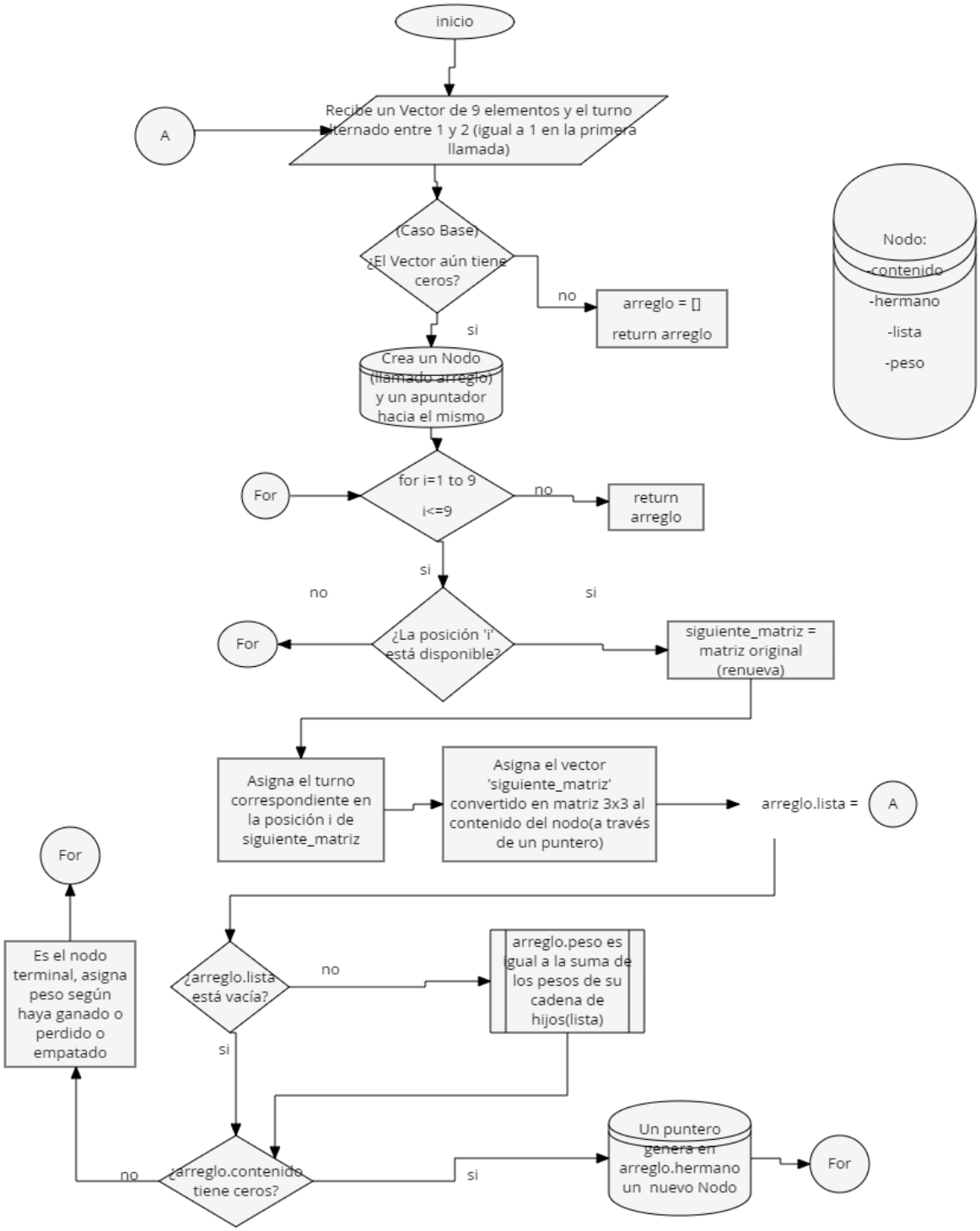
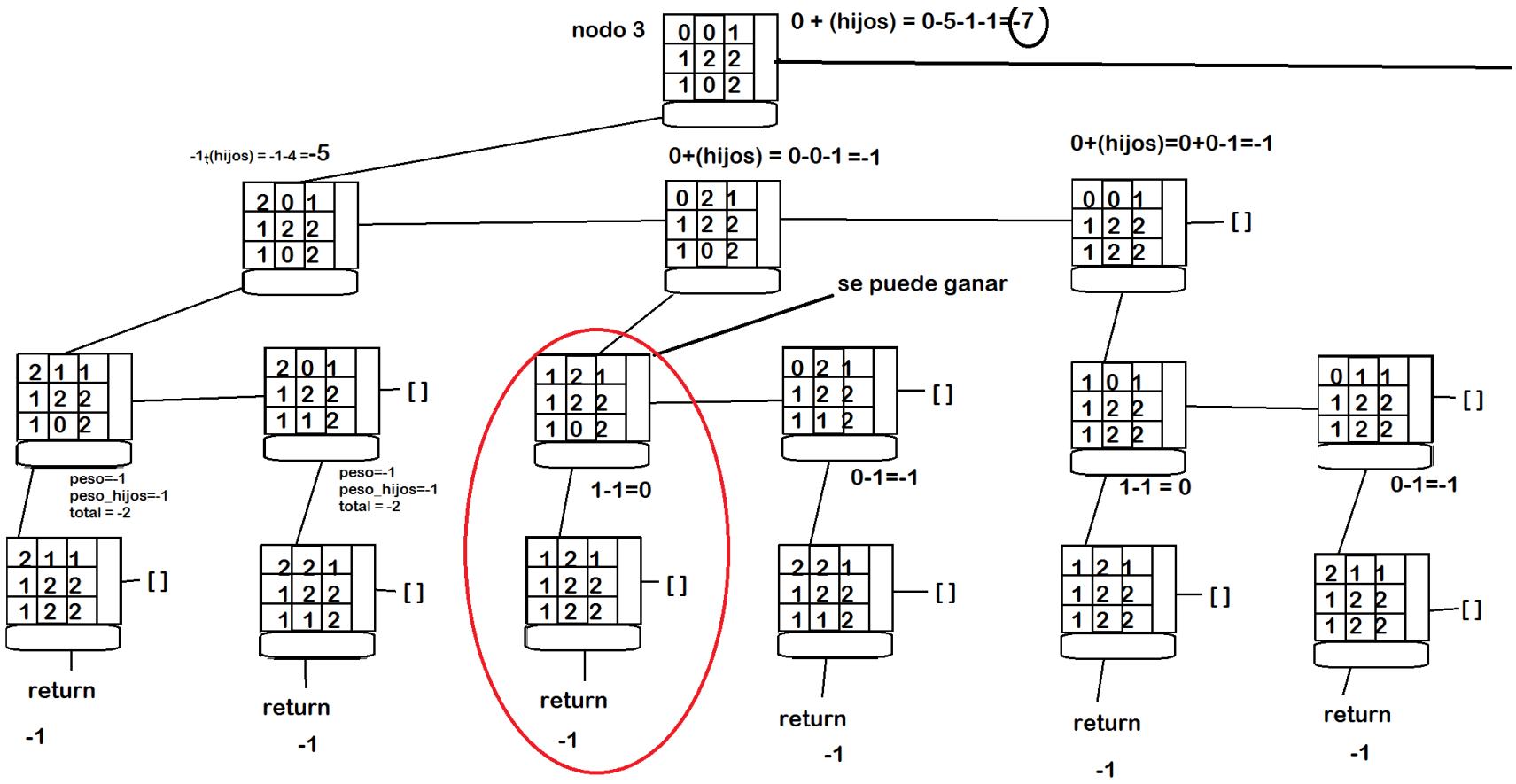


Diagrama 2, Diagrama de flujo de la función recursiva que genera todas las futuras posibles jugadas, (Es parte del diagrama de flujo principal-> Generar árbol de posibles jugadas)

Sin embargo al aplicar esa función, nos genera la siguiente ambigüedad (nos dice que perdimos, cuando en realidad ya ganamos mucho antes), por lo que se optó por corregir y de paso aumentar la eficiencia del algoritmo:



Función mejorada que genera todas las posibles jugadas del 'tres en raya' a partir de una matriz dada

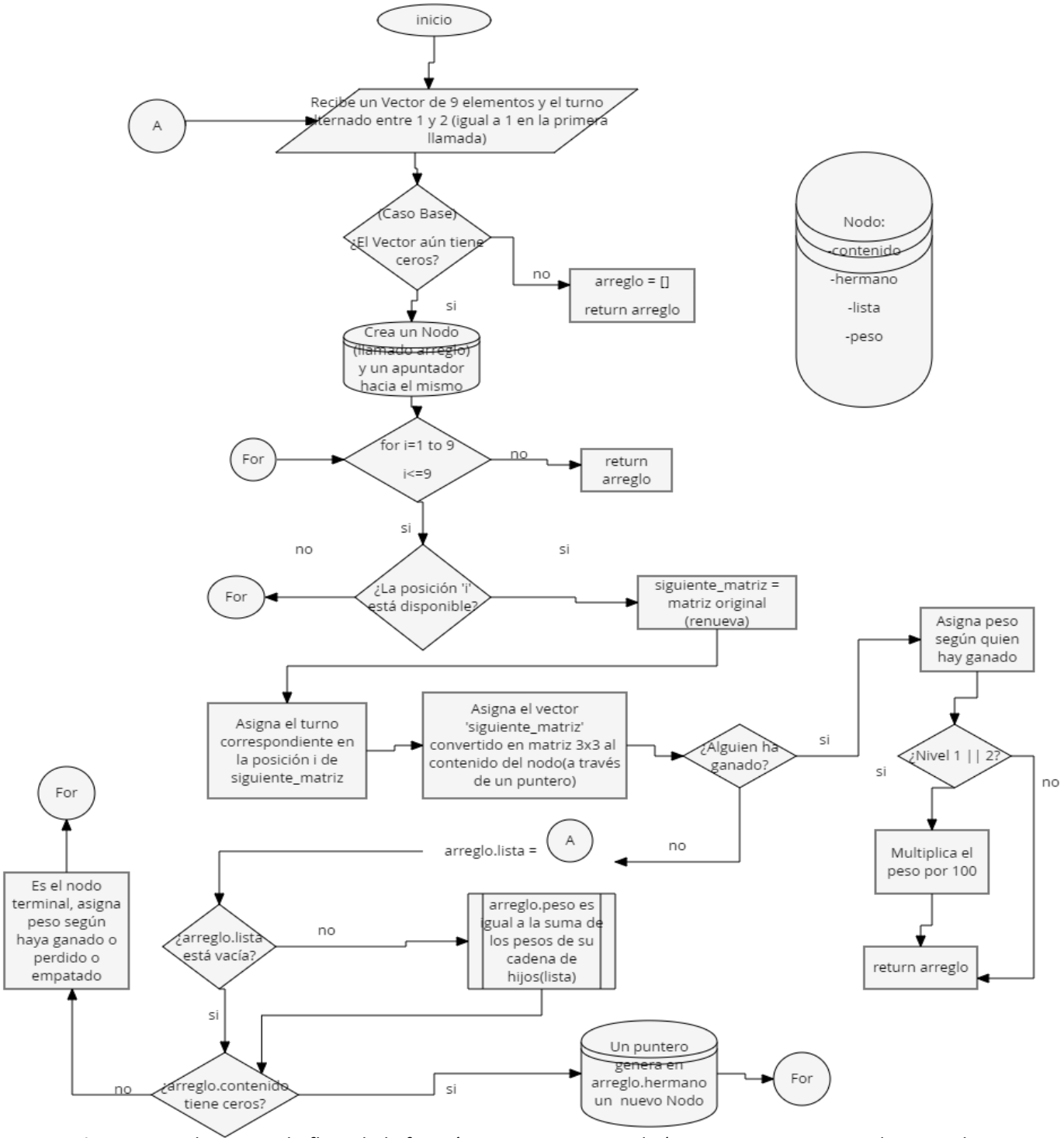


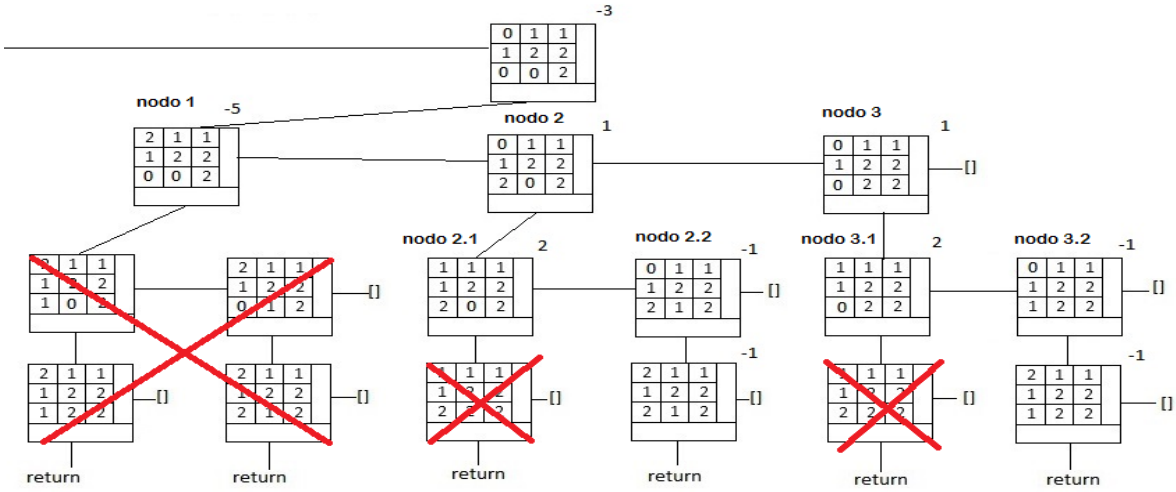
Diagrama 3, diagrama de flujo de la función recursiva mejorada (para ya no generar nodos cuando alguien de los dos ha ganado)

5. Estrategias

5.1 Estrategia en la asignación de pesos

Una vez generado el árbol hasta un punto terminal en el que alguien ganó (o se llegó a empate, por defecto) un valor positivo, neutro o negativo es asignado al peso de ese nodo: '1' cuando ganamos, '-1' cuando perdimos y '0' cuando empatamos, es importante notar que esos nodos terminales pueden estar en cualquier nivel de profundidad del árbol en el que se cumplan las condiciones ya nombradas.

Una vez asignado esos pesos de referencia, comenzamos a retornarlos al primer nivel de jugadas (que son justamente las posibles jugadas con las que podemos responder en ese momento) mediante el siguiente proceso:

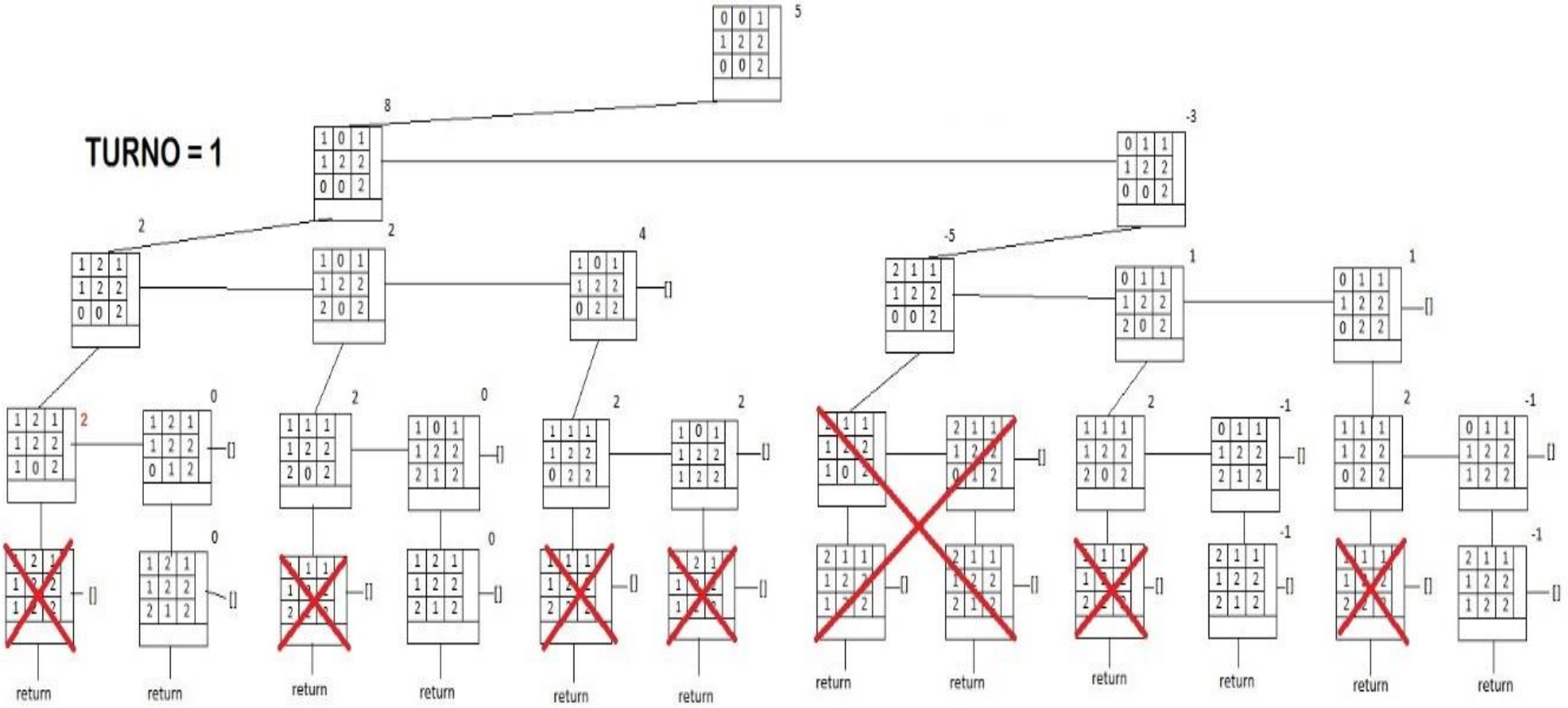


En esta figura nosotros empezamos la jugada ficticia ubicando un uno en la posición [1,2]. Y luego las jugadas se van alternando. En el nodo 1 apreciamos que hemos perdido por lo que se asigna un valor negativo, como fue un recorte y sus descendientes no fueron generados su valor es igual a su valor correspondiente '-1' más el número de nodos que recortó '4' (este '4' es negativo, es importante conservar el signo para no causar ambigüedad).

$$estado = -1;\ peso = estado + estado * (cantidad\ de\ nodos\ que\ debio\ generar) = -1 * -1(4) = -5$$

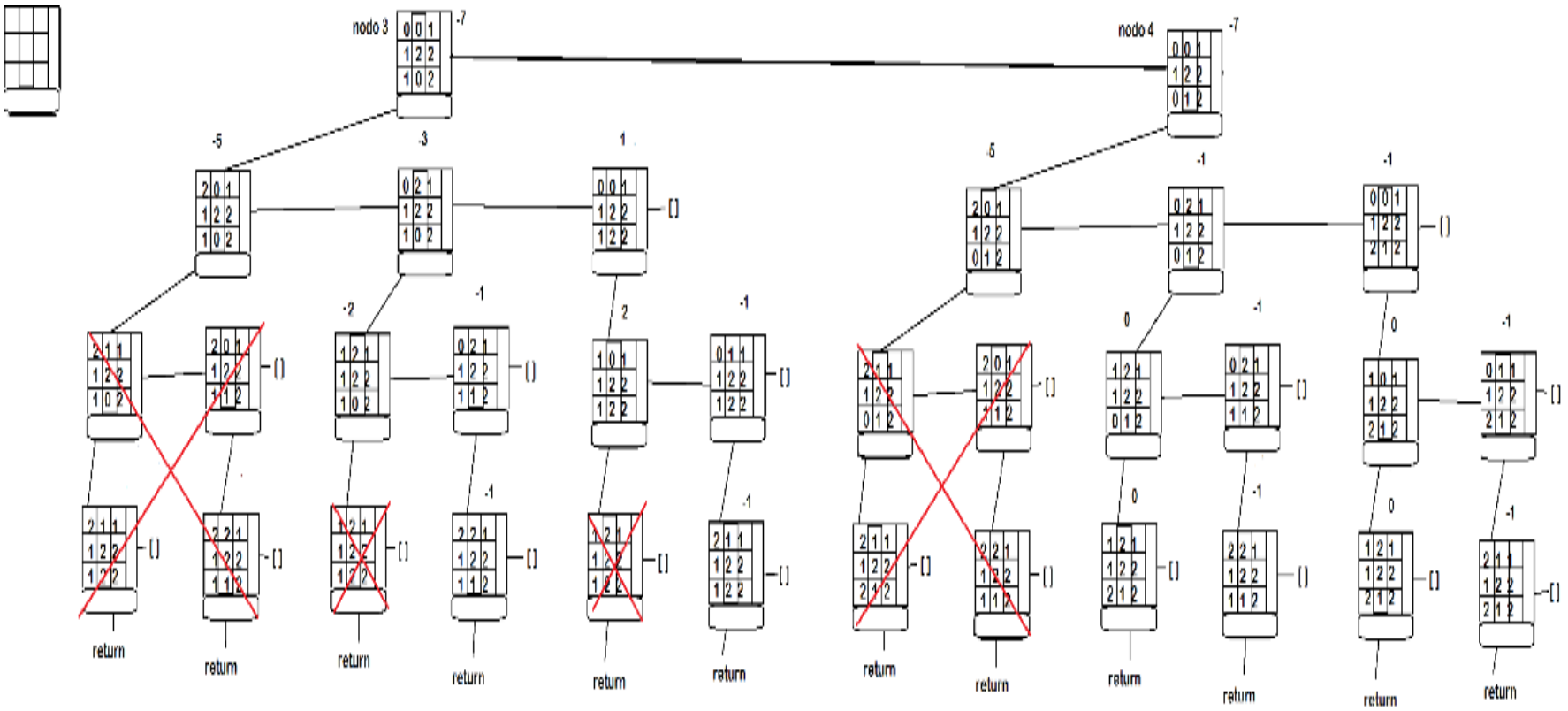
El número de nodos que debió generar se calcula mediante una fórmula obtenida de manera analítica que se explicará un poco más abajo de este documento. El nodo 2.1 y 3.1, realizan un proceso parecido. Para retornar el peso desde el nivel más profundo, se van realizando sumas de todos los nodos hijos en el anterior nivel, el peso del nodo 2.2 es igual a la suma del peso de todos sus nodos hijos (igual a -1), y el peso del nodo 2 es igual a la suma de los pesos de todos sus nodos hijos (2.1 y 2.2 -> 2 -1 = 1). Lo mismo para el nodo 3. Y de nuevo se realiza una suma entre el nodo 1,2 y 3 que da como resultado -3, y ese el peso de su pariente visible más 'viejo'.

Otro ejemplo de cómo se da la asignación de pesos:



Escuela Politécnica Nacional
Proyecto Tic Tac Toe

Hay 4 posibles jugadas, y los pesos nos indican que el nodo 1 con el peso de 8 es la mejor opción, y tiene razón.



5.1.1 Cantidad de nodos que genera la matriz con 'n' espacios disponibles

De manera analítica se obtuvo una fórmula para obtener la cantidad de nodos que cada padre generaba (esto es parte fundamental para la estrategia que usamos), para tres espacios disponibles (n=3) es:

$$[1 + (n-1) + (n-1)(n-2)] * n \ (uno)$$

Lo que se interpreta como: el primer descendiente (1) más el número de nodos que genera el primer descendiente (n-1) más el número de nodos que genera cada hijo (n-2) multiplicado por los hermanos de ese hijo (n-1) y como solo habíamos considerado el primer descendiente, debemos multiplicar por todos los demás descendientes del primer nivel (n). De manera general:

$$[1 + (1)(n-1) + \cdots + (1)(n-1)(n-2) \ldots (n-(n-1))] * n$$

Si desarrollamos la ecuación (uno):

$$= n + n(n-1) + n(n-1)(n-2)$$

$$= \frac{n(n-1)!}{(n-1)!} + \frac{n(n-1)(n-2)!}{(n-2)!} + \frac{n(n-1)(n-2)(n-3)!}{(n-3)!}$$

$$= n!\left(\frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{(n-3)!}\right) = n!\sum_{j=1}^{n}\frac{1}{(n-j)!} \ (dos)$$

Esta fórmula (dos) se reafirmó en el momento en que ubicamos un contador en el programa que aumentaba cada vez que se asignaba un contenido a un nodo generado. Con lo que de paso obtuvimos esta relación.

Tabla 1 – Comparación del tiempo que demora en generar los nodos con 'n' espacios

Espacios disponibles de la matriz (n)	Cantidad de nodos que genera (sin recortes)	Tiempo promedio de ejecución
1	1	0.0000
2	4	0.0003
3	15	0.0015
4	64	0.0100
5	325	0.0400
6	1956	0.2200
7	13699	1.5900
8	109600	13.3400
9	986409	90.7200

Sin embargo el árbol actualmente (con la implementación de recortar ramales innecesarios) no genera esa cantidad de nodos como se puede ver en la figura 2 (línea azul):

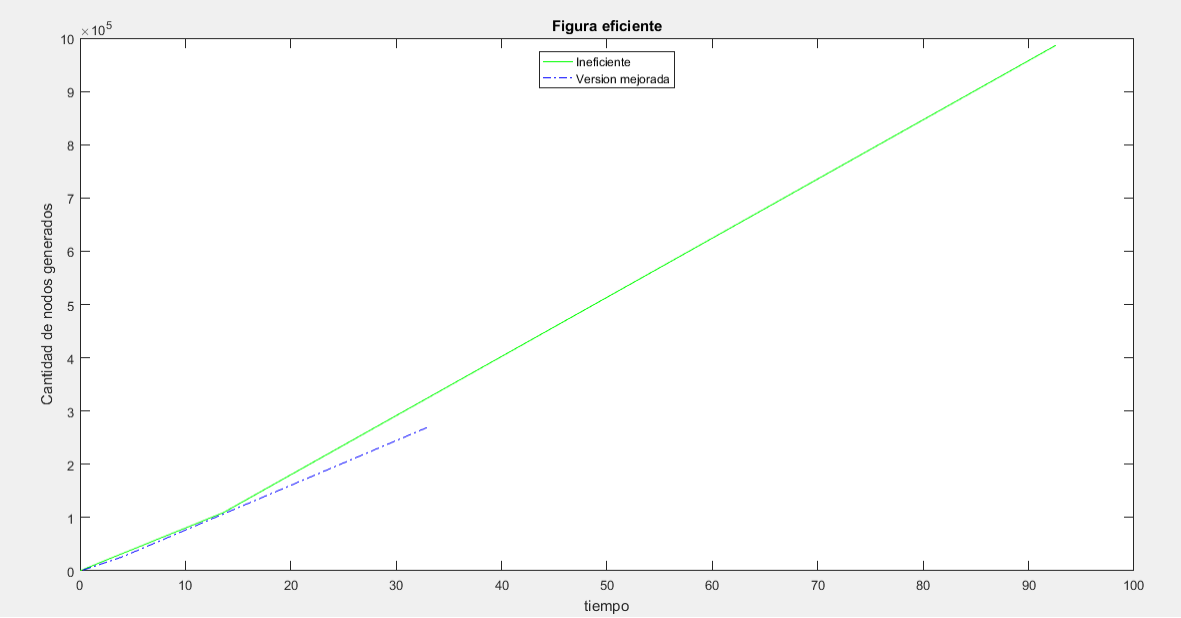


Figura 2, línea azul representa al programa recortando Nodos innecesarios, línea verde no recorta nodos innecesarios

Tabla 2 – Comparación entre el anterior y actual algoritmo

Espacios disponibles de la matriz (n)	Nodos que debe generar(línea verde)	Nodos reales que genera (línea azul)	Tiempo que debe tardar (línea verde)	Tiempo que se demora (línea azul)
1	1	1	0.0000	0.0000
2	4	4 aprox	0.0003	0.0003
3	15	15 aprox	0.0015	0.0015
4	64	43 aprox	0.0140	0.0120
5	325	130 aprox	0.0400	0.0250
6	1956	691 aprox	0.2200	0.0970
7	13699	3000aprox	1.5900	0.1400
8	109600	26852aprox	13.3400	4.0600
9	986409	269173aprox	90.7200	33.2800

La razón de 'aprox' es porque depende de si en la matriz de juego alguien está a punto de ganar o no, los valores que se obtuvo con matrices generales que terminaban en empate (peor caso) indicaban que se generaban al menos 0-1.5 veces menos que la cantidad real de nodos que debía generar. Ejemplo de matriz usada para contabilizar la creación de nodos.

```
[1 2 2;
 0 0 1; Esta matriz (n=4), generó exactamente 43 nodos en 0.012 segundos
 0 0 2]
```

5.2 Estrategia de la primera y segunda jugada

La 'Primera' y 'Segunda' jugada nacieron como estrategia para asegurar una victoria o al menos empate y porque resultaba más eficiente crear el árbol a partir de 7 espacios en blancos.

Según las reglas establecidas, el programa debe responder en menos de dos segundos, por lo que, si generamos el árbol con 9 espacios en blanco, perderíamos instantáneamente el juego.

La gráfica de espacios disponibles por llenar vs tiempo (ver figura 3) nos ayudó a determinar que con 7 espacios en blancos era suficiente para cumplir dicha regla.

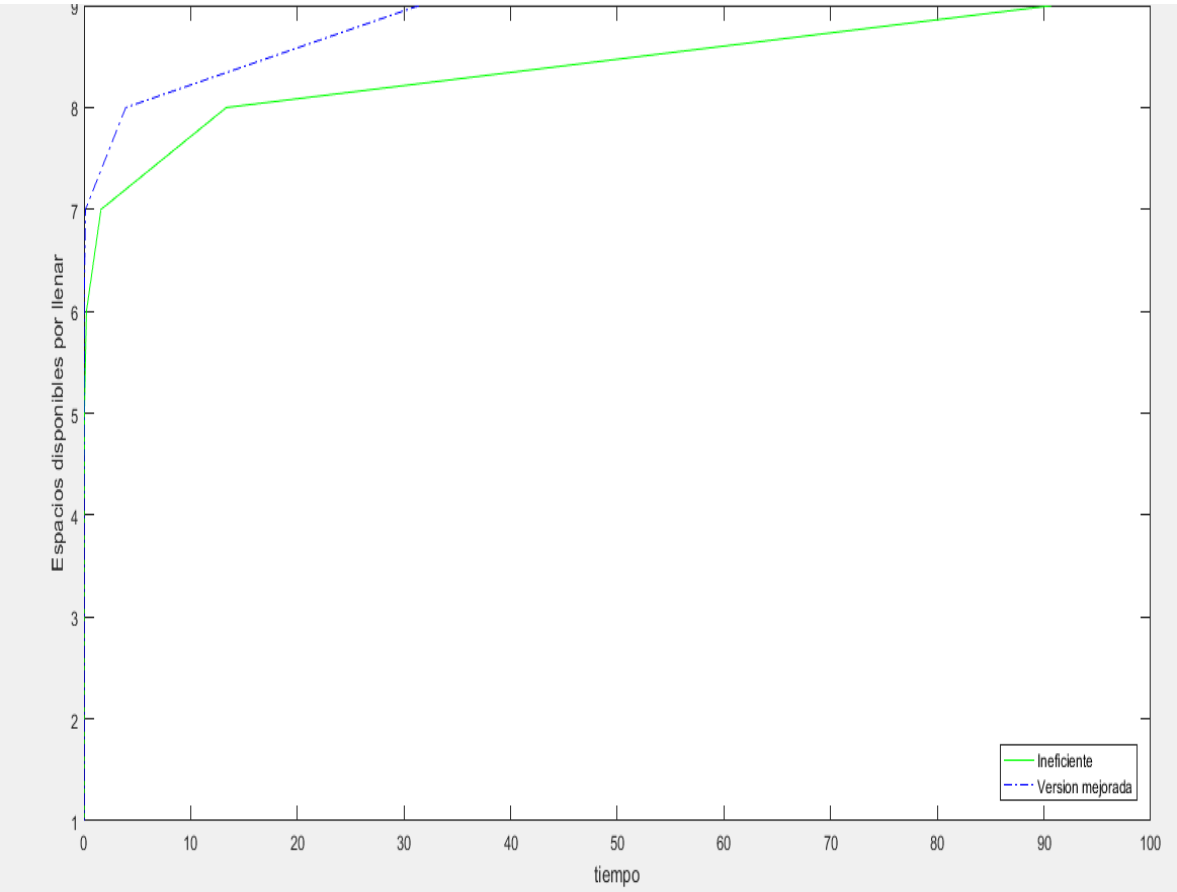
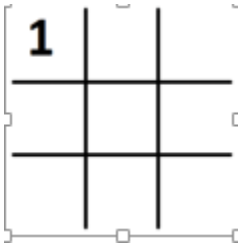


Figura 3, grafica de 'n' (cantidad de espacios disponibles por llenar) vs tiempo

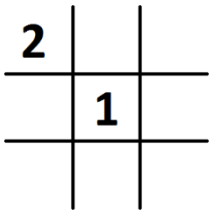
5.2.1 Primera Jugada:



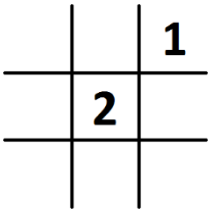
Los jugadores más experimentados colocan la primera jugada en una esquina cada vez que empiezan primero. Esto le da al oponente mayor oportunidad para cometer un error. Si el oponente responde colocando en cualquier parte que no sea el centro, las probabilidades de victoria aumentan. Es por ello que el algoritmo elige una de las 4 esquina como primera jugada.

5.2.2 Segunda Jugada:

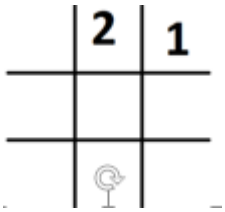
- Si el oponente juega primero y comienza en una esquina, siempre se coloca la segunda jugada en el centro. La segunda jugada no debe ubicarse en una esquina contraria, porque perderíamos automáticamente, sin vuelta atrás. Con esta estrategia, se estima que el juego terminará seguramente en empate. En teoría, se puede ganar desde esta posición, pero el oponente tendría que cometer un gran error.



- Cuando el oponente comience colocando en el centro, se coloca la segunda jugada en una esquina. Básicamente, no hay forma de ganar, solo de empatar, desde esta posición; a menos que el oponente cometa un error.

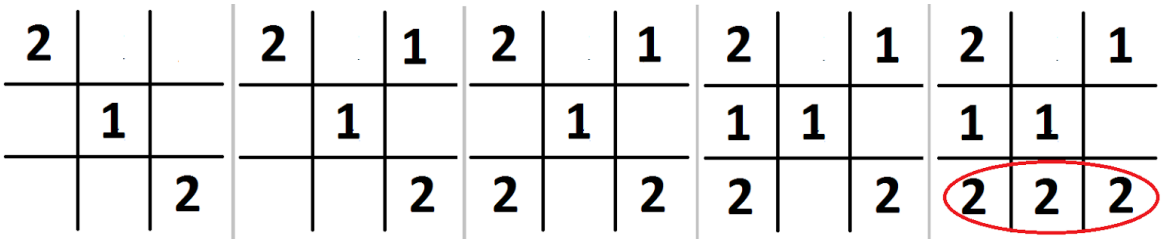


- Si el oponente coloca la primera jugada en un borde (lado) en lugar de en una esquina o en el centro, hay posibilidad de ganar. Se coloca la segunda jugada en una de las esquinas.

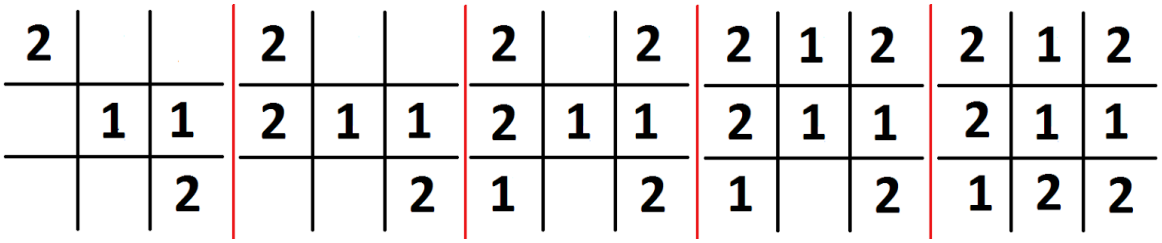


5.3 Contra-jugadas

La función implementada “contra Jugadas” sirve para solventar algunas debilidades que pueda tener el algoritmo y también para alimentarlo de “buenas jugadas”. Es decir, si encuentra alguna “jugada notable” en donde la victoria es clara, aplicamos esa jugada para ganar.
Por ejemplo:
Supongamos que tenemos esta jugada. Si colocamos un 1 en cualquiera de las esquinas sobrantes perderemos inmediatamente.



La ‘contra jugada’ está programada para ver casos simétricos de esta jugada con la función `rot90`, y ubicar su jugada en uno de los **lados**. Ya que si ubicamos en esas posiciones nuestro ‘1’ contrarrestamos esa jugada

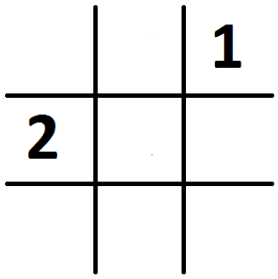


¿Por qué se da este problema?

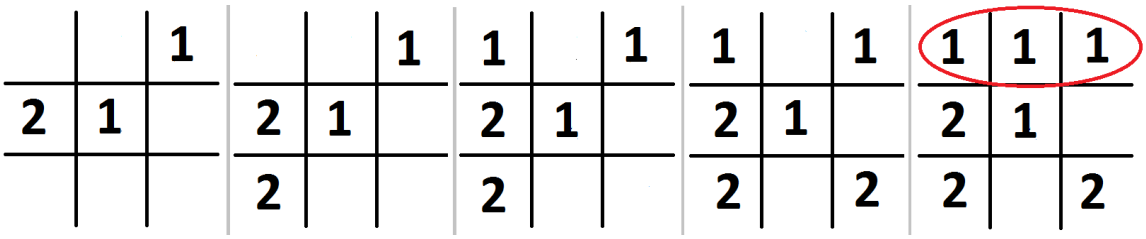
Porque los pesos nos pueden traer ambigüedad. El caso anterior es un ejemplo en donde la estrategia usada en el algoritmo genera ambigüedad.

Otro caso particular agregado en 'Contrajugadas' es el siguiente:

Suponga que tiene la siguiente matriz o algún caso simétrico



La maneara directa para ganar a partir de esa jugada es ubicando un 1 en el centro



¿Por qué un algoritmo 'casi excelente', provoca siempre empate contra un humano que sabe jugar (u otro algoritmo 'casi excelente')?

Esto se debe principalmente a que el juego del 3 en raya o tic tac toe es un juego resuelto, esto quiere decir que para todas las posibles jugadas que se pueden hacer para ganar en el juego existe al menos una jugada que puede evitarlo por lo tanto cada vez que alguien haga una jugada que le genere una ventaja el contrincante anulara su ventaja con otra jugada y así sucesivamente esto hace que el juego tienda a ser tablas (empate).

Por otro lado alcanzar estas tablas contra un algoritmo excelente o casi excelente se vuelve complejo para un jugador intermedio sin embargo al tratarse de un jugador que conoce bien el juego, tanto su concepto como estrategias, no se vuelve tan complejo entender la jugada del algoritmo y así realizar una contra jugada, por lo tanto el juego llegara a tablas.

Si se jugara una cantidad enorme de veces el resultado seguiría siendo tablas, ya que esto representaría el equilibrio entre el algoritmo y la persona (que domina el juego), aunque si llevamos el número de juegos muy arriba, la maquina seria el vencedor, no necesariamente porque juegue mejor sino por el error humano es decir después de varias partidas(cientos) el humano sentirá fatiga generada por realizar el mismo juego repetitivo una y otra vez sin embargo el algoritmo no tendría problema alguno en seguir , de esta forma seria el ganador, pero este caso es muy exagerado y no se ajusta a la realidad ya que también podríamos decir que al ser un algoritmo CASI perfecto va a tener algún limitante o alguna falla por lo tanto si el humano la descubre explotara este fallo para hacerse con la victoria, aunque esto no nos dirá quien juega mejor ya que para medir el mejor juego se buscara que ambos estén en las mejores condiciones.

Por otro lado el 3 en raya es un juego de ventaja, el jugador que más ventaja saque con sus movimientos será el que tenga mayor chance de ganar, pero volviendo al planteamiento de el algoritmo casi excelente y la persona que sabe jugar bien, entendemos que ninguno cometerá ningún fallo por lo tanto su rival no tendrá ninguna ventaja ambos realizaran lo que se conoce como un juego perfecto así q no habrá ventaja alguna que se consiga, entonces ¿Ninguno tiene ventaja?, parecería ser así sin embargo hay tener en cuenta que uno de los jugadores jugo primero así que dicho jugador ya tiene una ventaja ya que el tablero de 3 en raya tiene 9 espacio disponibles el que empiece primero podrá hacer 5 movimientos mientras que su rival solo tendrá 4 movimientos, en conclusión el primer jugador tiene ventaja pero ¿será suficiente esta ventaja? Como la mayoría que jugamos este juego nos dimos cuenta, no importa quien empiece primero si se realiza un juego perfecto el resultado seguirá siendo tablas, así que al final la ventaja que tiene el primer jugador no resulta suficiente es una ventaja que no decide el resultado del y es muy lógico ya que de ser al contrario bastaría con empezar primero y realizar un juego perfecto para ganar.

¿Qué pasaría si no existieran las tablas en el juego?

Si el empate no fuera una opción y los únicos posibles resultados fueran victoria o derrota, ¿ganaría el algoritmo o la persona? No se puede afirmar con certeza quien ganaría, pero si podemos afirmar que los resultados serían 100 a 0 es decir un jugador ganaría siempre y el otro perdería siempre, si ambos jugadores realizan un juego perfecto uno siempre ganara y otro siempre perderá, en este caso puede que la ventaja de ser el primer jugador sea decisiva en el resultado, por lo tanto uno siempre ganara y otro siempre perderá a pesar de que ambos realicen un juego perfecto.

Conclusiones:

- Esta estrategia y la de minimax pueden llegar generar ambigüedad, retornar un valor de 'cero' en muchos nodos aun cuando uno de ellos nos conviene elegir más que otro.
- La complejidad algorítmica de este programa está directamente relacionada con la cantidad de nodos que genera, en el peor caso sin la implementación el Big O, es de n^n(factorial).
- Modelar un árbol para las jugadas de una matriz de más altas dimensiones (como el tablero de ajedrez) resultará muy ineficiente computacionalmente.
- El resultado de tabla (empate) no es una casualidad sino un estado que debe pasar siempre que ambos jugadores (algoritmo o humano) realizan un juego perfecto, esto se debe a que el tic tac toe es un juego resuelto.

Recomendaciones:

- ✓ Es importante verificar si ganamos en el primer nivel de hijos generados, o si perdimos en el segundo nivel. En el caso de que ya ganamos en la siguiente jugada, debemos agregar peso extra muy alto a esa jugada y si perdemos en el segundo nivel, significa que no debemos elegir por nada, al nodo padre.
- ✓ Se debe utilizar estructura de datos para generar nodos encadenados a otros nodos de manera infinita (todas las posibles jugadas).
- ✓ Usar un controlador de versiones como 'Git' del que meridianamente se hablo en el apartado '3'.

Bibliografía:

- Tomás Blanco. *PARA JUGAR COMO JUGÁBAMOS*. 2003. ISBN 84-87339-44-1 Ed. John Wiley & sons.
- AlfonsoPinel , https://alfonsopinel.wordpress.com/
- Martin Gardner. Circo matemático. Alianza, 1995. ISBN 842061937X, 9788420619378
- (s.f.). Obtenido de https://unitorunozeydiio.files.wordpress.com/2011/03/quc3a9-es-un-c3a1rbol-de-decisic3b3n.pdf
- *ESTRUCTURA DE DATOS*. (Domingo 1 de Febrero de 2009). Obtenido de https://estructuradedatos-grupo1.blogspot.com/2009/03/lista-estructura-de-datos-en-ciencias.html
- *Estructuras de Datos*. (s.f.). Obtenido de https://hhmosquera.wordpress.com/arbolesbinarios/
- Huayna D., A. M. (s.f.). *Inteligencia Artificial 2012*. Obtenido de https://inteligenciaartificialfisi2012.files.wordpress.com/2012/04/mc3a9todos-de-busqueda-para-juegos-h-m.pdf
- Andrés. (2015 de 10 de 12). *codigofacilito*. Obtenido de Codigo Facilito Web site: https://codigofacilito.com/articulos/que-es-git
- Johnsonbaugh, R. (2005). Árboles de juegos. En R. Johnsonbaugh, *Matemática Discreta, sexta edición* (págs. 429-434). México: PEARSON EDUCACIÓN.