UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA
DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

Trabajo Final

GO

Análisis y Diseño de Algoritmos II

Hiese, Alan German
(alan.hiese@gmail.com)

Pitaro, Cristian Agustín
(agustin.pitaro@gmail.com)

ÍNDICE

Introducción

La propuesta de trabajo dada por la cátedra consistió en el diseño e implementación del juego de mesa llamado Go.
En el siguiente informe se procederá a detallar los aspectos del juego en general, las decisiones tomadas en base a la problemática planteada y el desarrollo de la solución propuesta.

¿Qué es el “Go”?

El “Go” es un juego de estrategia para dos personas creado en china hace más de 2500 años. El objetivo del mismo es intentar cercar territorios dentro del tablero del juego. El Go es rico en complejas estrategias a pesar de sus simples reglas. El juego es realizado por dos jugadores que alternativamente colocan piedras blancas o negras sobre las intersecciones libres del tablero, estos tableros poseen diferentes dimensiones, entre ellas las más populares son 19*19, 9*9.

El objetivo del juego es controlar una porción más grande del tablero que el oponente. Una piedra o grupo de estas, se captura y retira del juego si no tiene intersecciones vacías adyacentes, es decir, si se encuentra completamente rodeada por piezas del color contrario. Ubicar piedras juntas ayuda a protegerlas entre sí y evitar ser capturadas. Pero ponerlas separadas haces que tengas mayor influencia sobre más porción del tablero, entonces, hay que saber jugar con estas dos variables, haciendo que nuestro control sobre el tablero sea el mayor posible, pero que nuestras piezas están lo mas protegidas posibles.

Como curiosidad, se dice que los altos mandos americanos en la segunda guerra mundial, aprendieron a jugar al Go para tratar de entender los movimientos de las tropas japonesas (el Go es una disciplina militar obligatoria en Japón).

Como se juega?

Conceptos Básicos

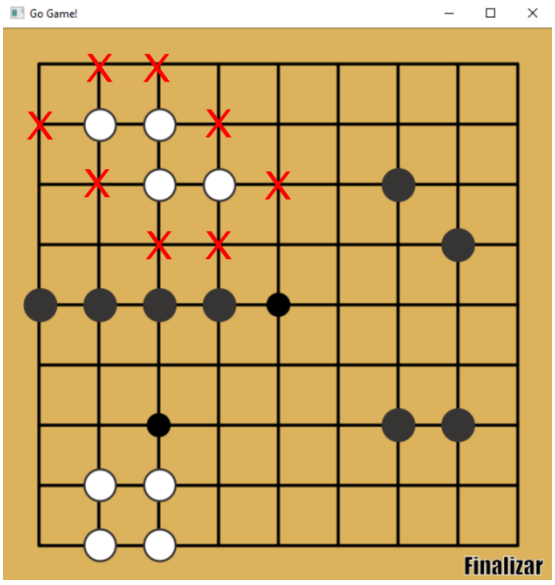
Un jugador utiliza piedras blancas y el otro, negras. Se turnan y ponen una piedra por vez sobre un punto vacío del tablero.

Cruces

Son todas las intersecciones del tablero.

Cadena

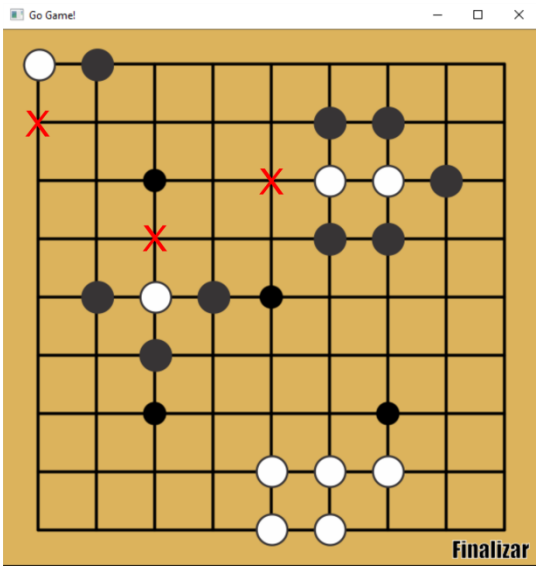
Cuando dos o más piedras del mismo color están unidas por líneas verticales u horizontales (no en diagonal), decimos que están conectadas formando una cadena.



En la figura hay dos cadenas blancas y cuatro cadenas negras.

La cadena blanca en la esquina superior izquierda tiene 8 libertades, marcadas con cruces.

Captura



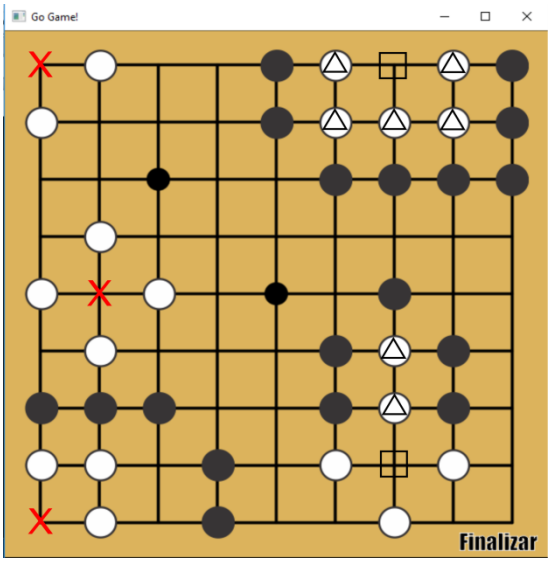
Una libertad es un cruce vacío adyacente a una piedra en el tablero.

Si luego de una jugada, una piedra o una cadena se queda sin libertades, esas piedras son capturadas por el oponente y retiradas del tablero.

Cuando a una piedra o cadena le queda una sola libertad, decimos que está en *atari*.

Si Negro juega en las casillas marcadas, captura las piedras blancas. Si Blanco juega en las casillas marcadas, gana libertades y evita ser capturado.

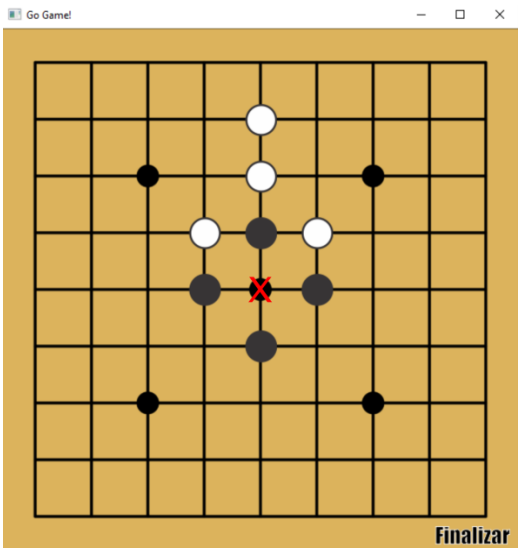
Suicidio



No se puede jugar una piedra de modo que quede sin libertades (suicidio). Sólo puede jugarse si en la misma jugada captura una piedra o cadena del rival.

Negro no puede jugar en las casillas marcadas con cruces, porque su piedra no tendría libertades. Pero sí puede jugar en las casillas marcadas con cuadrados, porque captura las piedras blancas marcadas con triángulos.

Regla del ko

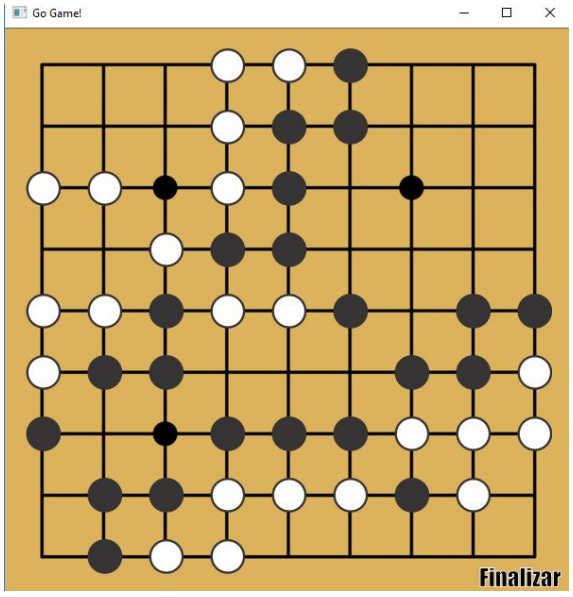


Quedan prohibidas las jugadas que repiten la posición del tablero de la jugada anterior.

Si Blanco juega en la casilla marcada, captura una piedra, pero su propia piedra queda en atari. Si luego Negro capturara, se repetiría la posición del diagrama, dando lugar a un ciclo infinito. La regla del ko prohíbe la repetición. Negro debe jugar en otro lugar.

Final de la partida

La partida termina cuando los dos jugadores están de acuerdo en que ninguno puede mejorar su posición, es decir, ganar más puntos.



Tanto Blanco como Negro han rodeado territorios. Para determinar el ganador, se cuentan los puntos de territorio de cada bando y se le suma un punto por cada piedra capturada durante la partida. El que obtiene más puntos es el ganador.

Negro tiene 36 puntos: rodeó 14 puntos de territorio y tiene 22 fichas en el tablero. Blanco tiene 33 puntos: 22 fichas en el tablero más 11 cruces dominados.

Esta forma de contar es en base en las reglas chinas.

Las reglas japonesas consisten en sumar las intersecciones que están bajo el control del jugador más el total de fichas capturadas, la diferencia entre estas y las otras reglas es que en las japonesas toda intersección del tablero tiene un jugador como dueño, mientras que en las reglas chinas puede que haya intersecciones que no pertenezcan a nadie. Otra diferencia es el hecho que si una ficha está rodeada por fichas enemigas aunque no esté en condición de ser capturada los jugadores pueden llegar a tomar la decisión de que tal ficha esté perdida (reglas japonesas)

Por estas diferencias es que se consideran las reglas chinas como más simples y sencillas para el aprendizaje y son estas las que se utilizan en el trabajo

Decisiones de diseño e implementación

Originalmente, la primera idea que se quiso llevar a la práctica era la implementación del juego mediante una matriz de enteros modelando así el tablero con sus respectivas fichas en él, y a su vez, enfocar la solución a la captura de terreno mediante cadenas de fichas (denominadas Cercos en su momento) salvo que a diferencia de las cadenas, estas tenían conexión con sus diagonales. Para implementar esta idea nos vimos en el inconveniente de que al centrarse nuestra solución en "¿Cómo capturar una ficha?", debíamos contemplar muchas situaciones que requerían, además de un complejo diseño del algoritmo en cuestión, una gran complejidad temporal a tal punto que solo se podía modelar un tablero de 5x5 con altos tiempos de ejecución.

El funcionamiento de dicho algoritmo era el siguiente, al colocar una ficha se buscaba en sus adyacentes fichas del mismo color y con esta nueva ficha entrabamos a un nuevo estado de recursión, de esta manera se trataba de llegar a la ficha de origen (contemplando límites del tablero incluso). El problema de este algoritmo no solo era el hecho del costo temporal de la recursión, si no que encontraba un mismo cerco repetidas veces, y cada uno de estos demandaba tiempo tanto encontrarlo, validarlo (porque una vez encontrado se tenía que validar que fuese un cerco real ya que había muchas combinaciones de fichas que no lo eran) y además una vez pasado todo ese proceso había que asegurarse que ese cerco no existiese en la lista de cercos del tablero.
Toda esta complejidad no solo era temporal, sino que el código era muy complejo de leer y debuguear, y como detalle final, los cercos no aportan nada a las reglas del juego o al puntaje final. Por lo que una solución sin estos era mucho más eficiente.

Al ver que esta forma de intentar resolver el problema tenía demasiadas limitaciones, se intentó buscar otra forma de modelar el problema y llegamos a la conclusión que debíamos centrarnos en otro punto importante en el juego, las cadenas. Esto simplificó en gran medida la complejidad general del trabajo pudiendo trabajar con tableros más grandes sin tanto contratiempo. En este trabajo se utilizan tableros de 5*5 y 9*9 aunque haciendo unas mínimas modificaciones puede extenderse a 19*19. Como dato extra, con la actual implementación, el tiempo de respuesta del programa en el modo *2 jugadores* es casi instantáneo tanto para un tablero de 5x5, 9x9 o cualquier otro que se presente, por lo que la mejora en este modo es significativa.

En cuanto a las cadenas, aunque en el juego original son de mínimo dos fichas, por cuestiones de implementación en el programa estas pueden estar compuestas por tan solo una ficha pero como el puntaje no se ve influenciado por la creación de cadenas esto no cambia el juego en si, tan solo es una herramienta para capturar fichas o un posicionamiento estratégico para el jugador.

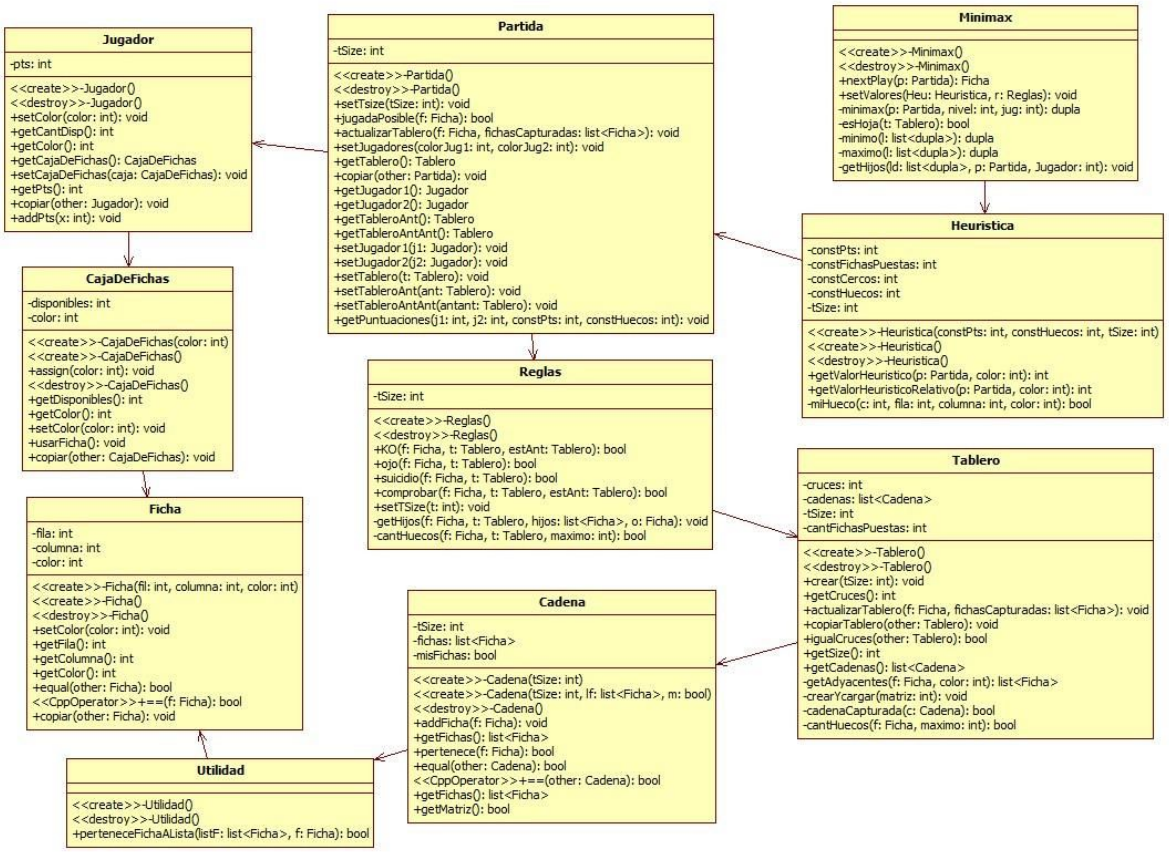
A continuación se detallaran la implementación de las principales estructuras usadas:

Tablero: Decidimos modelar el tablero como una matriz de enteros, donde cada entero simboliza la presencia o ausencia de una ficha (en caso de existir esta, se le da un valor para cada jugador). También fue necesario incluir una lista de cadenas junto con la cantidad fichas usadas. Vimos que era importante darle prioridad el acceso de cada lugar del tablero y que además tiene un tamaño estático, se decidió elegir la matriz ya que tiene O(1) para acceder a un casillero, en contra de hacer una lista de listas o una lista con una clase celda (la cual contendría la posición de la ficha).

Cadena: La cadena se modeló como una lista de Fichas, además de una matriz de booleanos que indica que territorio controla esa cadena en sí. Como era una estructura dinámica la que estábamos diseñando, la idea más directa que tuvimos fue implementarlo en Lista para así poder ir agregando y eliminando fichas en cualquier momento.

Ficha: El modelado de la ficha fue un caso simple ya que solo consta de valores de fila, columna y color de la misma por lo que bastó con diseñar una clase con dichos valores.

Diagrama de clases y explicación general



Se adjuntará al final del proyecto una versión de mayor tamaño de esta imagen.

Clases

Lo que representa cada clase es transparente con el nombre, con la salvedad de la clase *Utilidad* que su único funcionamiento es el de aportar una función de búsqueda en listas de fichas. Se creó esta clase en su momento ya que dicho comportamiento era utilizado en reiteradas ocasiones y en distintas clases, en la versión final del programa esta función sólo es utilizada por la clase *Cadena*.

Funciones

Solo las siguientes funciones son significativas, las demás son sencillas o son auxiliares al problema principal:

Funcion	Clase
getValorHeuristico	Heuristica
getValorHeuristicoRelativo	Heuristica
KO	Reglas
ojo	Reglas
suicidio	Reglas
comprobar	Reglas
nextPlay	Minimax
minimax	Minimax
jugadaPosible	Partida
actualizarTablero	Partida
actualizarTablero	Tablero

A continuación se detallan brevemente las funciones mencionadas anteriormente:

- **getValorHeuristicoRelativo:**

```
int Heuristica::getValorHeuristicoRelativo(Partida p, int color);
```

Dependiendo del color (que representa al jugador), retorna el puntaje de dicho jugador en relación al puntaje del otro.

- **getValorHeuristico:**

```
int Heuristica::getValorHeuristico(Partida p, int color);
```

Retorna el puntaje de un jugador, para obtener dicho valor lo único que hace es sumar los puntos del mismo y revisar cada cruce del tablero observando sus alrededores, dependiendo del estado de estos se decide si ese cruce pertenece o no al jugador (si en dicho cruce ya hay una ficha del jugador, éste ya pertenece a él sin importar sus alrededores).

Las tres funciones explicadas a continuación se basan en reglas del juego.

- **KO:**

```
bool Reglas::KO(Ficha f, Tablero t, Tablero estAnt) const{
    return t.igualCruces(estAnt);
}
```

Este algoritmo decide si una ficha produce un estado de KO . Su funcionamiento es simple, si el tablero actual es igual al de dos estados anteriores retorna *true*, en caso contrario retorna *false*.

•ojo:

```
bool Reglas::ojo(Ficha f, Tablero t) const;
```

Esta función retorna true si una ficha es puesta entre cuatro fichas enemigas, es una versión simplificada del suicidio, por lo que solo se fija en sus límites inmediatos.

• suicidio:

```
bool Reglas::suicidio(Ficha f) const{
    list<Cadena> cadenas = t.getCadenas();
    typename list<Cadena> :: iterator itC = cadenas.begin();
    while (itC != cadenas.end()){
        if (itC->pertenece(f))
            break;
        itC++;
    }
    list<Ficha> hijos = itC->getFichas();

    typename list<Ficha> :: iterator itLista = hijos.begin();
    while (itLista != hijos.end()){
        Ficha fA(itLista->getFila(),itLista->getColumna(),itLista->getColor());
        if (!cantHuecos(fA,t,0))
            return false;
        itLista++;
    }
    return true;
}
```

Para confirmar si una ficha está en estado de suicidio o no, esta función encuentra la cadena a la que pertenece f (siempre hay una, en el caso mínimo es una cadena compuesta solamente por esa misma ficha) y luego retorna *false* si alguna de las fichas que la componen tiene al menos una libertad.

•comprobar:

```
bool comprobar(Ficha f, Tablero t, Tablero estAnt) const;
```

En esta función retorna *true* o *false* dependiendo de los valores que retornen las tres funciones anteriores, sabiendo así si es correcto poner la ficha *f*.

•nextPlay:

```
Ficha nextPlay(Partida p);
```

Esta es la función pública del minimax, solamente se encarga de llamar a este ya que es privado y de retornar al usuario la ficha a jugar por la máquina.

•minimax:

```
dupla minimax(Partida p, int nivel, int jug){
	/*dado que el algoritmo original es demasiado largo, se explicará mediante
	un pseudocódigo*/
	if (estadoHoja())
		list<dupla> Q = generarValoresHeuristicos();
		return mejor(Q);
	else{
		for todas las fichas f posibles{
			nivel++;
			Partida pAux = p;
			cambiarJugador(jug);
			If (!poda)
				dupla d = Minimax(pAux,nivel,jug);
			Q.add(d);
		return mejor(Q);

	}
}
```

En esta función, si es estado hoja (tablero lleno o imposibilidad de movimientos para ese jugador) se agrega a la lista Q todas las posibles fichas con sus valores heurísticos y se retorna la mejor opcion segun el jugador (el minimo si es el jugador humano y el máximo si es la máquina). En caso de que no sea estado hoja se lleva a cabo la recursión para todas las fichas posibles y de manera similar a lo anterior se selecciona la mejor ficha con la salida de la función recursiva (en la implementación se utiliza un if para separar a ambos jugadores pero es solo para una mejor legibilidad).

•jugadaPosible:

```
bool Partida::jugadaPosible(Ficha f) {
        if (f.getColor() == 0){
            if (j1.getCantDisp()>0)
                return r.comprobar(f,t,tAntAnt);
        }
        else {
            if (j2.getCantDisp()>0)
                return r.comprobar(f,t,tAntAnt);
        }
    return false;
}
```

Esta función es la encargada de informarle al usuario si una jugada es posible, para esto hace uso de las reglas y de la cantidad de fichas disponibles del jugador.

•actualizarTablero (Partida):

```
void Partida::actualizarTablero(Ficha f, list<Ficha> &fichasCapturadas){
    tAnt.copiarTablero(t);
    tAntAnt.copiarTablero(tAnt);
    t.actualizarTablero(f,fichasCapturadas);
    if (f.getColor() == 1){
        j1.getCajaDeFichas().usarFicha();
        j1.addPts(fichasCapturadas.size());
    }
    else{
        j2.getCajaDeFichas().usarFicha();
        j2.addPts(fichasCapturadas.size());
    }
}
```

Actualiza los tableros auxiliares anteriores, el tablero actual, y al jugador le actualiza tanto los puntos como las fichas usadas.

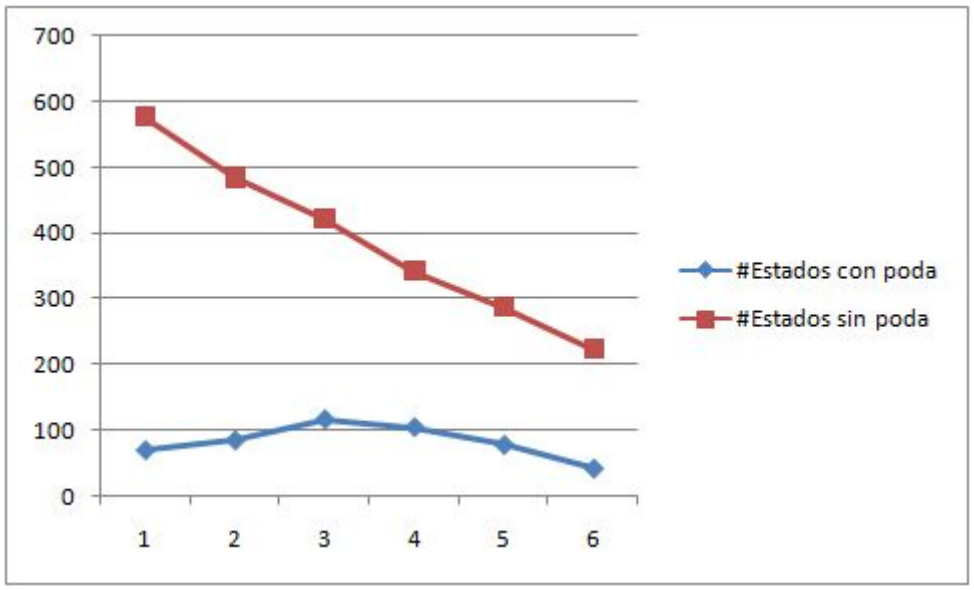
• actualizarTablero (Tablero):

```
void Tablero::actualizarTablero(Ficha f, list<Ficha> &fichasCapturadas){
	/*al igual que en la función minimax, se presenta un pseudocódigo*/
	cruces[f.getFila()][f.getColumna()] = f.getColor();
	bool sinCambios = true;
	Para toda Cadena c{
		if (c.pertenece(f)){
			c.add(f);
			sinCambios = false;
		}
	}//si hay más de una cadena a la que pertenezca f, las une
	if (sinCambios){
		Cadena c;
		c.add(f);
		c.add(getAdyacentes(f));
		cadenas.add(c);
	}
	if (!tableroLleno){
		Para toda Cadena c{
			if (c.getLibertad == 0){
				cadenas.remove(c);
				fichasCapturadas.add(c.getFichas());
				//actualizar matriz con -1 dónde estaban las fichas de
	c
			}
		}
	}
}
```

Esta función actualiza el tablero en todos sus aspectos, como primer cosa, actualiza la matriz, luego en el primer *Para* agregar a *f* a todas las cadenas a las que pertenezca, y en caso de no pertenecer a ninguna crea una nueva con ella y sus adyacentes en caso de que los tenga. En el otro *Para* elimina (solo si el tablero no esta completo) todas las cadenas que se quedaron sin libertad, añade a fichasCapturadas las fichas que componen estas cadenas y actualiza la matriz.

Análisis computacional

Para llevar a cabo algunas mediciones y poder compararlas entre sí y además que estas sean significativas, lo que se hizo fue poner 6 fichas consecutivas negras (la máquina respondía dependiendo la ficha colocada), dichas fichas son las mismas en ambos casos y la prueba está hecha en un tablero de 5x5. Los resultados fueron los siguientes:



Luego de la 6ta ficha en la prueba sin poda, el programa deja de responder por memoria insuficiente.

Como se puede observar, el cambio entre cada uno de los programas es más que significativo, aunque es verdad que en el programa sin poda (función roja) se nota una descendencia más abrupta a medida que se colocan más fichas, la cantidad de estados sigue siendo muy superior a la del programa con poda (función azul), el cual va reduciendo la cantidad de estados de una forma mucho más gradual.

Luego de analizar ambos experimentos y con el dato de que luego de la 6ta ficha el programa sin poda deja de responder debido a memoria insuficiente decidimos quedarnos con el programa que implementa una poda.

Para llevar a cabo dicha poda, lo que se hizo fue colocar antes del llamado recursivo en la función *minimax* una condición que evita dicho llamado. Para esto utiliza una variable local llamada poda (a la que inicialmente se le asigna un valor extremo dependiendo si es jugada del usuario o de la máquina) y solo se va a hacer el llamado recursivo en caso de que el valor heurístico obtenido para el tablero actual, de un puntaje para ese jugador mejor que el almacenado en la poda. Este método evita la gran mayoría de intentos recursivos que generan valores iguales en el juego temprano, este tipo de poda posee el riesgo de no analizar a una mayor profundidad la jugada, por lo que hay casos en lo que una jugada que si avanza da como resultado un mayor puntaje, es ignorada por tener el mismo puntaje que una evaluada anteriormente, aun así se decide mantener este método por la gran eficiencia que aporta.

Interfaz gráfica

Para resolver dicha cuestión, se decidió elegir Simple and Fast Multimedia Library (SFML). SFML es una API portable, escrita en C++ pero también disponible en C, Python, Ruby, OCaml y D. Su propósito principal es ofrecer una biblioteca alternativa a la biblioteca SDL, usando un enfoque orientado a objetos.

Gracias a sus numerosos módulos, SFML puede ser usada como un sistema mínimo de ventanas para interactuar con OpenGL o como una biblioteca multimedia cuyas funcionalidades permiten al usuario crear videojuegos y programas interactivos.

La misma puede ser accedida por medio de su pagina web: www.sfml-dev.org en la cual se pueden encontrar desde su descarga hasta cursos y un foro para dilucidar cualquier duda de la misma.

Conclusion

El Go es un juego que es altamente jugado en el mundo, en especial Japón y sus alrededores. Es una lastima que Argentina no sea tan difundido ya que aunque sea un juego de reglas simples, encierra una gran complejidad estratégica. Por esa misma razón, en Japón, es tan avalado por la sociedad, habiendo incluso escuelas superiores de este deporte.

Durante el desarrollo de este proyecto fuimos testigos de esto, sobretodo cuando abordamos el modelado del jugador artificial, ya que tuvimos que tomar muchas consideraciones para limitar el backtracking y así reducir el número de posibilidades intentando penalizar lo menos posible el resultado final de dicho jugador.

Como conclusión final podemos decir que, aunque fue agotador, fue una experiencia muy gratificante el ver dicho juego diseñado e implementado por nosotros.