Jugador virtual del Go, basado en el algoritmo de Monte Carlo

Marcos Pividori
Docente: Ana Casali
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Resumen. En este trabajo se presenta el desarrollo de un jugador virtual del Go, basado en Monte-Carlo Tree Search (MCTS). Inicialmente se desarrolla una librería general, adaptable y eficiente para el algoritmo MCTS, con múltiples ejemplos de uso en distintos dominios. Luego, se procede a trabajar en el problema particular del juego Go, introduciendo mejoras principalmente en la etapa de simulación a través de la incorporación de conocimiento de dominio. En particular, se implementan mejoras a través de la detección de patrones en el tablero y la consideración de múltiples movimientos claves en el juego. También, se exponen diferentes decisiones en el diseño del programa haciendo énfasis en una plataforma eficiente y reusable. Finalmente se presentan los resultados obtenidos del jugador desarrollado en comparación a otros programas alternativos.

1 Introducción

Como trabajo final de la materia Introducción a la Inteligencia Artificial, se decidió implementar un programa que juegue al Go. Revisando el estado del arte (por ej. [1] [5] [6] [9] [12]), se puso en manifiesto que el enfoque principal que se utiliza hoy en día para enfrentar este problema son los programas construidos sobre el algoritmo de Monte Carlo Tree Search [5].
El Go ha sido considerado siempre un gran desafío para la Inteligencia Artificial. Uno de los mayores obstáculos para la construcción de un programa, ha sido la dificultad de conseguir una adecuada función de evaluación. Para hacer frente a esta dificultad, se presenta el algoritmo MCTS, que permite evaluar una posición simulando un conjunto de partidas partiendo de dicha posición y analizando los resultados obtenidos al final.
En este informe, inicialmente se exploran las principales alternativas independientes del dominio de aplicación, desarrollando una librería general para el algoritmo MCTS. Luego, se procede a trabajar en el problema particular del Go, investigando las mejoras disponibles en el estado del arte, principalmente a través de la incorporación de conocimiento de dominio en la etapa de simulación. Se implementan varias de ellas, considerando las más significativas en el rendimiento, y de esta manera se construye un programa de cierto nivel competitivo sobre el que se pudo investigar e implementar diferentes mejoras propias. Finalmente se presentan resultados del jugador desarrollado en comparación a otros programas alternativos.

2 MCTS

Monte Carlo Tree Search (MCTS) es un método para toma óptima de decisiones en problemas de Inteligencia Artificial. Combina la generalidad de simulaciones aleatorias con la precisión de una búsqueda en el árbol de posibilidades. MCTS no requiere una función de evaluación de posición, en contraste con la búsqueda alfa beta. Está basado en una exploración aleatoria del espacio de búsqueda, pero usa los resultados de previas exploraciones. Para ello, MCTS construye gradualmente un árbol en memoria, que mejora sucesivamente estimando los valores de los movimientos más prometedores.

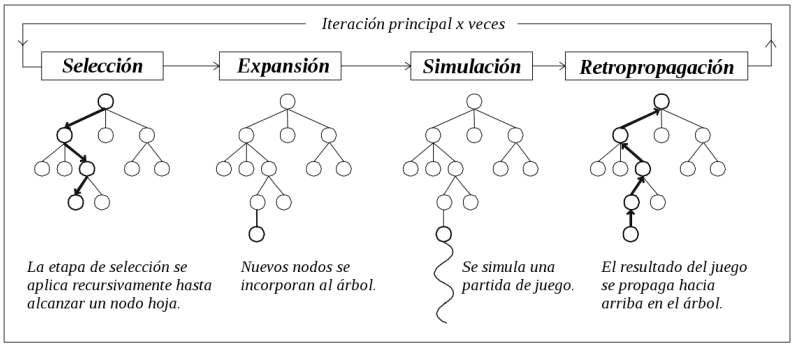

Figura 1: Etapas del algoritmo MCTS [5].

MCTS consiste en cuatro etapas principales, repetidas tantas veces como tiempo se disponga. En cada una de las iteraciones se parte de la situación actual del juego.

- **Selección:** El árbol se recorre desde el nodo raíz hasta alcanzar un nodo hoja. Se toma una rama u otra dependiendo de la estrategia de selección que se emplee y la información almacenada en el nodo en ese momento, como el valor y el número de visitas.

Dentro de las diferentes opciones, UCT (Upper Confidence bounds applied to Trees) [4] es la más utilizada por su simplicidad y eficiencia. La estrategia UCT calcula para cada uno de los movimientos posibles una combinación de dos valores, la tasa de éxito de ese nodo y la relación entre el número de veces que se ha visitado el nodo padre y el número de veces que se visitó dicho nodo (hijo). El primer valor está relacionado con la *explotación* y el segundo con la *exploración*. A través de un coeficiente C utilizado en la combinación de ambos valores, se puede dar mayor prioridad a la explotación o exploración. Un punto importante a favor de la estrategia UCT, es su independencia del dominio de aplicación.
UCT: $ValUCT(N_i) = tasaExito_i + C * \sqrt{ln(n_p)/n_i}$
Donde: n_p y n_i son el número de visitas al nodo padre y al nodo N_i respectivamente, C es el coeficiente de exploración.

- **Expansión:** Se añaden nodos al árbol MCTS según una estrategia de expansión. Según el criterio, se puede expandir siempre que se visite un nodo, o cuando se alcanza un determinado número de visitas mínimo, lo que permite ahorrar espacio en memoria, regulando cuánto crece el árbol en relación al número de ciclos del algoritmo. Además, se pueden tomar dos caminos a la hora de expandir: en cada expansión añadir un solo nodo hijo de todos los posibles movimientos o añadir directamente todos los nodos hijos.
- **Simulación:** Se simula una partida a partir del nodo hoja alcanzado en las fases anteriores. Durante esta partida, el programa juega solo, realizando los movimientos de todos los jugadores que intervienen hasta que finalice y se obtenga un resultado. Las estrategias que se utilizan consisten o bien utilizar movimientos aleatorios o combinar la aleatoriedad con una heurística asociada al problema concreto. En estos casos es necesario buscar un equilibrio entre la exploración, que da la aleatoriedad, y la explotación, que dirige hacia un movimiento más prometedor.
- **Retropropagación:** El resultado de la simulación se propaga hacia los nodos atravesados previamente. Partiendo del nodo hoja y subiendo por la relación con los nodos padres hasta llegar a la raíz, se actualiza cada nodo, incrementando en una unidad el número de visitas y actualizando su valor con el resultado de la simulación.

Por último, a la hora de seleccionar el movimiento final, se considerará el mejor hijo del nodo raíz, es decir, el movimiento más prometedor de acuerdo a la información recaudada. Para determinar qué nodo "es mejor" se pueden tomar diferentes criterios, considerando la tasa de éxito, o el número de visitas, etc.

3 Implementación MCTS

Para la implementación del algoritmo MCTS, principalmente se buscó:

- Que sea *reusable*. Es decir, que la implementación del árbol y el algoritmo MCTS sean independiente del dominio de aplicación, y además se puedan agregar nuevos módulos para modificar diferentes etapas del algoritmo (Selección, Expansión, Simulación, etc.), en caso de tomar diferentes criterios.
- Que sea *eficiente*. Este punto siempre estuvo presente en la implementación. Para lograr buenos resultados, es vital realizar la mayor cantidad de ciclos del algoritmo en el menor tiempo posible. Por esto, se intentó optimizar en todo lo posible, y en muchas casos se recurrió al uso de templates, funciones *inline*, guardar cachés de ciertos valores para evitar hacer cálculos redundantes, sobre todo de funciones costosas (por ejemplo las llamadas a log() y sqrt() en las selecciones uct y rave), etc.

Buscando un código reusable, se decidió abstraer toda la lógica del dominio de aplicación en una clase *State* y un tipo *Data*. La clase *State* abstrae las formas en que puede transicionar el sistema, todas las reglas de juego, el cálculo de puntajes, los posibles pasos a tomar en un determinado punto, etc. El tipo *Data*, almacenará los datos de una transición a realizar en un determinado momento, por ejemplo, en el caso del tateti, *Data* almacenará las coordenadas de la ficha a incorporar y si es cruz o círculo. Luego, *State* contará con una interfaz mínima:

```
class State{
    State(State *src)
    get_possible_moves(vector<Data>& v)
    apply(Data)
    Value get_final_value()
}
```

La cual se implementará para cada dominio en particular. Por ejemplo: *StateTateti*, *StateGo*, *StateConnect4*, etc. Cada nodo del árbol almacenará información *Data*, de un movimiento determinado. Entonces, partiendo de un estado inicial, es posible ir recorriendo el árbol desde la raíz, aplicando los pasos que se encuentran en cada nodo hasta llegar a una hoja, donde se iniciará la simulación. De esta manera se evita almacenar un estado en cada nodo, almacenando únicamente los datos de la transición realizada (*Data*).
Para las 4 etapas principales: *Selección*, *Expansion*, *Simulación*, *Retropropagación*, se crearon clases abstractas con las interfaces mínimas necesarias para llevar a cabo el algoritmo.
Diferentes implementaciones que corresponden a diferentes criterios, pueden heredar de cada una de ellas, como por ejemplo: *SelectionUCT*, implementará la selección utilizando el algoritmo UCT, mientras que *SelectionUC-*

TRave, incorporará además la evaluación *amaf* siguiendo la propuesta de optimización RAVE.

4 Paralelización

Con respecto a la paralelización del algoritmo, se consideraron diferentes opciones, de igual manera a las mencionadas en [3]. Luego de hacer algunas pruebas con diferentes enfoques, se decidió proseguir con el principio de Tree Parallelization [3] con un lock global. En este enfoque, cada thread bloqueará el árbol cuando necesite acceder al mismo, en las etapas de selección, expansión y retropropagación. Puede parecer muy limitante, pero como el mayor tiempo se pierde en la etapa de simulación, este criterio termina siendo muy válido y se pueden lograr resultados muy buenos con una implementación muy simple, solamente de debe agregar un lock global para el acceso al árbol.
Como segunda opción, se implementó el enfoque Root Parallelization, debido a que se mencionaban muy buenos resultados en la documentación [3]. Root Parallelization consiste en mantener un árbol distinto para cada thread que actúa de forma independiente al resto y al finalizar, se unen en un solo árbol sobre el que analiza el mejor movimiento. Esto permite el máximo de concurrencia, ya que cada thread actúa de forma independiente, pero al costo de tener mas información almacenada en memoria. En nuestro caso, no se lograron mejoras en el tiempo de ejecución ni en la porcentaje de partidas ganadas, por lo que se prosiguió con el enfoque Tree Parallelization. Sin embargo, Root Parallelization es soportado y se puede activar a través de una bandera.

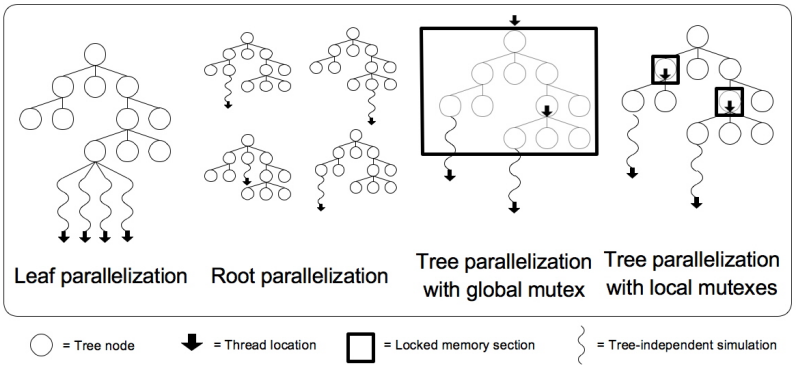

Figura 2: Diferentes enfoques de paralelización MCTS. [3]

5 Algunas pruebas

Para testear la implementación del algoritmo MCTS con algunos ejemplos simples, se implementaron diferentes juegos. En cada caso se prosiguió de manera similar, encapsulando la lógica del juego en una clase State y añadiendo la posibilidad de que el usuario pueda interactuar jugando a través de la consola.
Las estrategias utilizadas en cada etapa fueron:

- **Selección:** UCT.
- **Expansión:** Luego de 2 visitas, se expanden todos los hijos posibles.
- **Simulación:** Movimientos totalmente aleatorios.
- **Retropropagación:** Simple, actualizando las tasa de éxito y num. de visitas.
- **Selección del resultado:** Más Robusta, que consiste en elegir el nodo hijo con mayor número de visitas.

Tateti: Se presenta un tablero de 3x3 y cada paso consiste en el ingreso de una ficha en una posición vacía. El objetivo será lograr una linea de 3 fichas iguales (Diagonal/Vertical/Horizontal).
Connect4: Se presenta un tablero de 6x7 y cada paso consiste en el ingreso de una ficha en una columna con al menos una posición vacía. El objetivo será lograr una linea de 4 fichas iguales (Diagonal/Vertical/Horizontal).
Hexa: Se presenta tablero de 11x11 y cada paso consiste en el ingreso de una ficha en una posición vacía. El objetivo será lograr una cadena de fichas iguales que conecten ambos extremos del tablero. A diferencia de los dos ejemplos anteriores, en este caso el factor de crecimiento del árbol $(11 * 11 = 121)$ es mucho mayor. Para poder hacer frente a esto, se recurrió a la mejora RAVE (7), logrando mucho mejores resultados.

6 El juego Go

El Go es un juego de tablero estratégico para dos jugadores. Durante el juego, los jugadores se alternan para colocar piedras sobre las intersecciones vacías de una cuadrícula o pasando (no colocando ninguna piedra). Negro mueve primero. Una vez colocada una piedra, no se mueve por el resto del juego. Una piedra o cadena de piedras del mismo color es capturada y retirada del juego si después de una jugada, se encuentra completamente rodeada por

piedras del color contrario en todas sus intersecciones directamente adyacentes. El objetivo del juego es obtener la mayor puntuación al final.
Existen diferentes formulaciones de las reglas del juego, pero todas concuerdan en los aspectos generales y las diferencias no afectan significativamente la estrategia ni el desarrollo del juego salvo en situaciones excepcionales. El juego termina después de 2 pases consecutivos, cuando ningún jugador piensa que existan más movimientos convenientes y por lo tanto deciden no mover (Pasar).
A pesar de que las reglas de Go son simples, la estrategia es extremadamente compleja e involucra balancear muchos requisitos, algunos contradictorios. Por ejemplo, ubicar piedras juntas ayuda a mantenerlas conectadas y si una está viva, también lo estarán las demás. Por otro lado, colocarlas separadas permite tener influencia sobre una mayor porción del tablero con la posibilidad de apropiarse de más territorio. Parte de la dificultad estratégica del juego surge a la hora de encontrar un equilibrio entre estas dos alternativas. Los jugadores luchan tanto de manera ofensiva como defensiva y deben elegir entre tácticas de urgencia y planes a largo plazo más estratégicos.

6.1 Implementación

El programa se desarrolló principalmente en el lenguaje C++, debido a la necesidad de un entorno eficiente que a la vez permite combinar los paradigmas de programación genérica y orientada a objetos resultando en un código estructurado en clases, a la vez muy adaptable a las diferentes aplicaciones a través del uso de templates.
El código se encuentra disponible online en: `www.github.com/MarcosPividori/Go-player` como software libre bajo licencia GNU GPL3 [17].

1. **Go Text Protocol (GTP) [16]:** es un protocolo basado en texto para la comunicación con programas que juegan al Go. A través del intercambio de comandos, se puede establecer partidas entre diferentes programas, programas y usuarios y conectarse con servidores para participar de torneos.
   El proyecto desarrollado, soporta este protocolo. Esto nos permitió testear los diferentes avances estableciendo partidas contra otros programas, como *GnuGo* [14], y diferentes variantes de si mismo.
   Al mismo tiempo, también nos posibilitó comunicarnos con una interfaz visual, como ser *GoGui* [15], pudiendo así trabajar en un entorno más amigable, analizando el programa durante su desarrollo.
2. **La clase StateGo:** Para registrar el estado del juego, se recurre a una matriz que almacene las posiciones del tablero (Blanco/Negro/Vacío). Por otro lado, es necesario representar de alguna manera la noción de "Bloque". Para esto, se buscó una implementación que permita realizar las operaciones sobre ellos lo más eficientemente posible. Las principales operaciones y su complejidad de acuerdo a la implementación elegida son:
   - Saber si el bloque está rodeado por fichas del color opuesto. O(1)
   - Saber a qué bloque pertenece una ficha en una posición. O(1)
   - Agregar fichas a un bloque. O(1)
   - Unir 2 bloques. O(n) (n tamaño del menor bloque)
   - Eliminar del tablero las fichas de un bloque. O(n) (n tamaño del bloque)

7 Mejora Rapid Action Value Estimation

El algoritmo MCTS calcula por separado el valor de cada estado y acción en el árbol de búsqueda. Como resultado, no se puede generalizar entre las posiciones o movimientos relacionados. Para determinar el mejor movimiento, muchas simulaciones deben realizarse a partir de todos los estados y acciones.
Para hacer frente a esta dificultad, el algoritmo RAVE [2] propone utilizar la heurística "all-moves-as-first" (AMAF) con el objetivo de calcular un valor general para cada movimiento, independientemente del momento en que se realiza. Esta mejora está basada en la característica de ciertos juegos incrementales, como el Go, donde el valor de una acción a menudo no está afectada por movimientos realizados en otro lugar en el tablero.
De esta manera, la heurística AMAF proporciona órdenes de magnitud más información: cada movimiento típicamente se ha intentado en varias ocasiones, después de sólo un puñado de simulaciones, resultando en una estimación rápida del valor de cada acción y una mejora significativa en el rendimiento del algoritmo de búsqueda.
Sin embargo, aunque el algoritmo RAVE aprende muy rápidamente, a menudo es equivocado. La principal asumpción de RAVE, que un movimiento en particular tiene el mismo valor en todo un sub-árbol, se viola con frecuencia. Este problema se supera mediante la combinación del aprendizaje rápido del algoritmo RAVE con la precisión y la convergencia del algoritmo MCTS. Para estimar el valor total de la acción a en un estado s, se utiliza una suma ponderada, entre el valor *amaf* y el valor *mc* en dicho nodo: $Q(s,a) = (1-p)*Q_{\text{MC}}(s,a)+p*Q_{\text{amaf}}(s,a)$
Donde p se irá modificando a medida que se realizan mayor número de simulaciones. Valiendo $p(N_i) \approx 1$ cuando n_i (num de simulaciones) es un número menor y un valor $p(N_i) \approx 0$ cuando se realizaron un número considerable de simulaciones n_i. Si además se incorpora el componente de *exploración* de la selección UCT, se obtiene una nueva fórmula para el algoritmo de selección:
UCT-RAVE[2]: $ValUCTRave(N_i) = (1 - p(N_i)) * tasaExito_i + p(N_i) * amaf_i + C * \sqrt{ln(n_p)/n_i}$

Donde: n_p y n_i son el número de visitas al nodo padre y al nodo N_i respectivamente, C es el coeficiente de exploración y p es el coeficiente que determina la relación entre la estimación amaf y la tasa de éxito.
Se decidió definir el valor de p de acuerdo al enfoque "Hand-Selected Schedule" [2], el cual utiliza un parámetro k que determina el número de simulaciones en el cual p asignará igual peso a la *tasa de éxito* y a la estimación *amaf*: $p(N_i) = \sqrt{k/(3 * n_i + k)}$
Esta mejora se implementó a nuestro algoritmo de MCTS de la siguiente manera:

- Incorporando una nueva clase SelectionUCTRave que hereda de la clase abstracta Selection e implementa la funcionalidad antes descripta.
- Fue necesario modificar los nodos del árbol para llevar cuenta de los valores amaf.(NodeUCTRave)
- Creando una clase MoveRecorder que lleve el registro de los movimientos realizados en una simulación. (y por lo tanto actualizando el proceso de simulación para que utilice esta clase).
- Modificando el algoritmo de retropropagación para que actualice adecuadamente los valores amaf de los sub árboles de acuerdo a los movimientos registrados en la simulación.

Nuevamente, se buscó una implementación independiente del dominio de aplicación.

8 Mejora en las estructuras de datos utilizadas

Intentando mejorar el número de simulaciones realizadas por tiempo, se analizaron diferentes optimizaciones sobre la clase StateGo. En cierto punto, se hizo evidente un cuello de botella en el proceso de obtener la lista de movimientos posibles en cada instante de juego. Para esto, se recorría toda la matriz del juego, buscando posiciones libres, que no impliquen suicidios ni otras restricciones de juego (como ser Ko). Esto llevaba a un costo fijo en cada iteración de la simulación. Para evitar este costo, se propuso llevar un registro de los movimientos disponibles en cada instante de juego. Las principales operaciones sobre este conjunto serían:

- Insertar un movimiento.
- Eliminar un movimiento.
- Obtener el número de movimientos disponibles.
- Acceder al movimiento ubicado en la posición i.

Los dos últimos puntos se deben a la necesidad de elegir un movimiento aleatoriamente. Para esto es necesario poder enumerarlos y acceder a ellos una vez elegido un número aleatorio dentro del conjunto.
Sin embargo, de las estructuras esenciales disponibles, no se contaba con ninguna que permita realizar todas las operaciones en una complejidad aceptable. Por ejemplo, se analizaron:

- List: para insertar o eliminar un elemento de manera única, requiere un costo lineal, al igual que para obtener el elemento i-ésimo.
- Vector: permite obtener el elemento i-ésimo en tiempo constante. Pero insertar y eliminar elementos de manera única tendrá un costo lineal.
- Set: permite insertar y eliminar elementos en tiempo log(n), pero acceder al elemento i-ésimo tendrá un costo lineal. Es decir, se permite un acceso secuencial y no aleatorio a los elementos.

Por lo tanto, se decidió implementar una nueva estructura que permita realizar las 3 operaciones en una complejidad aceptable. Se comenzó con un árbol binario AVL [13], continuamente balanceado, que asegura complejidad log(n) para inserción y eliminación de elementos.
Para permitir acceso aleatorio a los elementos, se decidió incorporar una modificación sobre el árbol, donde cada nodo lleva un registro del número de elementos en el sub árbol izquierdo, es decir, el sub árbol que contiene los elementos menores. Esta modificación incorpora gastos mínimos en la actualización del árbol, pero permite buscar el elemento i-ésimo con una complejidad log(n).

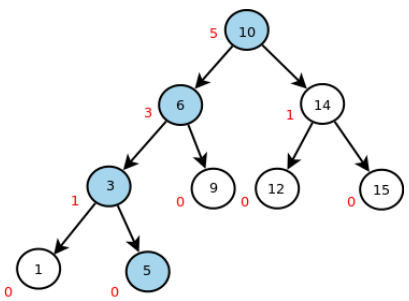

Figura 3: Ejemplo de búsqueda del elemento en la posición 2. Número rojos representan el número de nodos en el subárbol a izquierda de cada nodo.

9 Incorporación de conocimiento de dominio

Con el objetivo de mejorar el algoritmo, y habiendo implementado las principales opciones independientes del dominio de aplicación, se decidió considerar algunas alternativas a partir de la incorporación de conocimiento propio del juego Go.
Las principales etapas sobre las que se puede actuar son las de *Selección* y *Simulación*:

- **Selección:** existen varias mejoras posibles, como ser:
  - Progressive Bias [7]: Consiste en guiar la búsqueda incorporando heurísticas en la etapa de Selección, las cuales tendrán gran influencia cuando el número de simulaciones es menor y a medida que el número aumenta, su aporte disminuirá hasta ser nulo.
  - Progressive Widening [7]: consiste en podar el árbol en relación al tiempo disponible y número de simulaciones realizadas, con el objetivo de reducir el factor de crecimiento y concentrar la búsqueda en las mejores opciones encontradas hasta el momento.
- **Simulación:** La simulaciones totalmente aleatorias resultan en partidas sin mucho sentido, perdiendo utilidad a la hora de evaluar un movimiento. La idea intuitiva es que las simulaciones deben ser los más cercanas a un juego de la realidad, de manera de poder evaluar con cierta seguridad como proseguiría el juego si se realizan determinados movimientos iniciales. Entonces, resulta un desafío lograr un correcto balance entre exploración y explotación. Las simulaciones no deben ser demasiado aleatorias, ya que llevarían a resultados sin mucho sentido, ni demasiado deterministas, ya que perderían un gran rango de posibilidades que tome el juego. Para lograr esto, dentro de la documentación propia del área, se encontraron 2 principales enfoques:
  - Urgency-based simulation (Bouzy [5]): En cada instante de la simulación, un valor de urgencia U_j es computado para cada movimiento j posible, combinando un valor "capture-escape" (que considera el número de fichas que serían capturadas y el número que lograrían escapar de una captura, con dicho movimiento) con un valor que se calcula buscando ciertos patrones de 3x3 en el tablero. Luego, cada movimiento será elegido con mayor o menor probabilidad de acuerdo a su valor de urgencia U_j.
  - Sequence-like simulation (Gelly [8]): Consiste en seleccionar ciertos movimientos de interés en un área cercana al último movimiento realizado, resultando en una secuencia de movimientos cercanos uno del otro.
    Para seleccionar un movimiento, se buscan respuestas locales a través de patrones. Estos patrones, son matrices de 3x3, centradas sobre una intersección libre (que representa el próximo movimiento a realizar) que intentan dar respuestas a algunas situaciones clásicas del juego Go. Son considerados únicamente en el perímetro alrededor del último movimiento realizado.
    Los fundamentos detrás de este enfoque son:
    - Probablemente, movimientos cercanos serán la respuesta apropiada para contrarrestar los últimos movimientos realizados.
    - Es más importante obtener mejores secuencias de movimientos que mejores movimientos aislados.

A pesar de encontrar varios enfoques para introducir conocimiento, no resulta claro cómo evaluar cual es mejor sin hacer pruebas uno mismo con cada uno de ellos. Probablemente, el mejor resultado se logre a partir de una combinación de diferentes enfoques.
De esta manera, inicialmente se comienza mejorando la etapa de Simulación a través del uso de secuencias de movimientos, porque:

- No requiere un gran gasto computacional como es el caso de calcular valores de urgencia para todos los movimientos en cada instante de simulación.
- Es bastante simple de implementar.
- Es el enfoque utilizado en MoGo[8], el cual logró muy buenos resultados siendo programa pionero en el área.

Luego, se incorporó la noción de movimientos claves como *"Capturas"* y *"Escapes de Atari"*, que no necesariamente deberían ser cercanos a los últimos movimientos realizados. Y también otras mejoras disponibles en el estado del arte como los movimientos *"fill board"* [10] para explorar áreas no visitadas del tablero. Todos estos movimientos se combinaron apropiadamente a través de un algoritmo que da más o menos prioridad dependiendo de la importancia de cada uno.

1. **Patrones:** Los patrones utilizados en nuestras pruebas, fueron tomados de [9], y corresponden a varias situaciones clásicas del juego Go, como se muestra en Fig. 4.

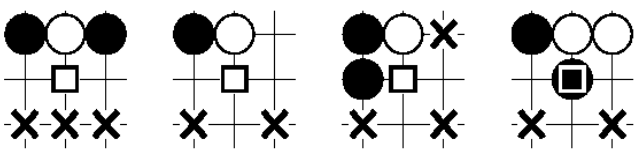

Figura 4: Patrones para la situación Hane [9]. "X" representa cualquier situación.

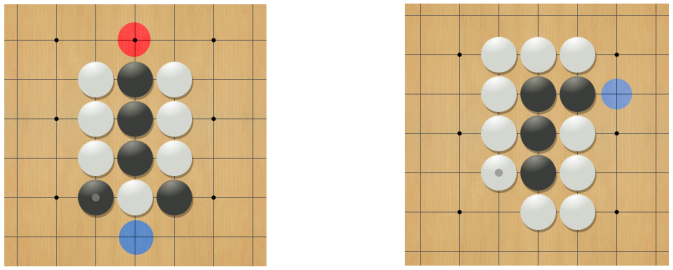

Figura 5: A izquierda: posiciones de escape de atari a través de una captura de un bloque opuesto. La posición azul significará una captura de blancas y escape de atari del bloque negro principal, que pasará a tener dos adyacencias libres. De igual manera la posición roja para las fichas blancas. A derecha: simple posición de escape de atari, sin capturar bloques enemigos.

2. **Capturas y escapes de atari:** Además de las posiciones que concuerdan con patrones, otras posiciones resultan de gran importancia en el juego, las posiciones que determinan cuándo los bloques son capturados.
   Por lo tanto, se incorporan algoritmos para detectar los movimientos de *"Captura"*, es decir, los que permiten capturar bloques del oponente y los movimientos *"Escape de atari"*, que permite aumentar el número de adyacencias de un bloque propio en estado de *atari* (una sola intersección libre y por lo tanto propenso a ser capturado por el oponente), como se puede ver en la Fig. 5.
   Para evitar calcular dichos movimientos en cada instante de la simulación, lo cual impondría un gran costo computacional, se opta por llevar un registro de los bloque en estado de *atari* durante el juego. De esta manera, se podrán calcular:
   - *Movimientos de escape de atari:* considerando los bloque propios en estado de *atari*.
   - *Movimientos de captura de bloques oponentes:* considerando los bloques del oponente en estado de *atari*.
3. **Algoritmo:** Fue necesario modificar el algoritmo de simulación para reemplazar la elección de movimientos puramente aleatorios por una elección que dé mayor importancia a las posiciones antes mencionadas.
   Para elegir qué movimiento tomar en cada instante de la simulación, basándose en las propuestas presentadas en [10] y [6], se probaron diferentes variantes y finalmente se concluyó con el siguiente algoritmo:

```
list = []
Por cada movimiento de ESCAPE DE ATARI Mi que salva al bloque Bi,
 incorporar size(Bi) veces Mi a list y si Mi además captura un
 bloque enemigo Ei, incorporarlo size(Ei) veces más.
Si size(list) < CTE:
 Intentar encontrar aleatoriamente una posición "FILL BOARD".
 Si no se encuentra:
  Incorporar a list toda posición adyacente (8-adj) al último
   movimiento que concuerde con algún PATRÓN.
  Por cada movimiento Mi de CAPTURA al bloque Bi, incorporarlo
   a list size(Bi) veces.
  Si list está vacía:
   Incorporar a list todos los movimientos posibles.

Elegir, de forma uniformemente aleatoria, una posición de list
 y realizar dicho movimiento.
```

   De esta manera, se incrementa la probabilidad en que son elegidos los movimientos de mayor importancia. Es decir, un movimiento que me permite evitar que un bloque de tamaño 10 sea capturado, probablemente será mucho más importante que un movimiento que concuerde con un patrón, o uno que solo capture un bloque de tamaño 1. A la vez, si tenemos dos movimientos que nos permiten escapar de una situación de *atari*, probablemente sea de mayor interés aquel que a la vez captura fichas del contrincante, incorporando en cierto sentido el concepto de "Urgencia" (Bouzy [5]).
   Sin embargo, aún si se incrementa la probabilidad de elegir ciertos movimientos, no se anula el resto, de manera de asegurar cierta aleatoriedad en las simulaciones que permita obtener una estimación más abarcativa del rumbo que pueda tomar el juego.
   Cuando se menciona una posición "fill board", se refiere a la mejora mencionada en [10], que permite considerar ciertos movimientos en áreas no exploradas del tablero.
4. **Duración de las simulaciones:** Las partidas pueden llevar muchos movimientos, derivando en simulaciones muy largas. Sin embargo, en la mayoría de los casos, el resultado final simplemente se puede estimar después de un cierto número de movimientos, donde la partida ha llegado a un punto en que ya hay un ganador que difícilmente cambiará. De esta manera, cortando el juego en este punto y evaluándolo, permitirá lograr

simulaciones mucho más cortas y por lo tanto poder realizar muchas más en igual tiempo.
Sin embargo, tampoco resulta adecuado cortar las partidas muy tempranamente pues la estimación podría ser errónea. Por lo tanto, se modificaron las simulaciones para poder establecer un límite de movimientos como: $umbral = coeficiente * num\_celdas\_del\_tablero$, donde el coeficiente se puede establecer, al igual que muchos otros, a través de linea de comando y tendrá un valor por defecto de 3. Por ejemplo, en un tablero de 9x9, se limitará a 243 movimientos.
Estos movimientos se cuentan por simulación, de manera que al iniciar el juego, cuando se han realizado muy pocos pasos, las simulaciones terminarán relativamente temprano. Pero a medida que avance el juego, las simulaciones cada vez llegarán más lejos, refinando de esta manera los resultados estimados.

5. **Selección del mejor movimiento:** Cuando se juega partidas contra otro jugador, muchas veces se llega a un punto en que resulta muy poco probable poder ganar, y por lo tanto, no tiene sentido seguir jugando, cuando en realidad se podría aprovechar este tiempo para una nueva partida. Por lo tanto, se modifica la etapa de Selección del resultado para que, una vez elegido el mejor movimiento estimado por MCTS, si este movimiento representa una probabilidad de ganar por debajo de un coeficiente, se opte por *pasar*. (De esta manera, si el contrincante está en ventaja también pasará y terminará el juego)

6. **Implementación:** Esta mejoras se implementaron a nuestro algoritmo de MCTS de la siguiente manera:
   - Incorporando una nueva clase *SimulationWithDomainKnowledge* que hereda de la clase *Simulation* e implementa el algoritmo para la selección del paso a realizar en cada instante de la simulación. También incluirá el límite antes mencionado sobre la duración de las simulaciones.
   - Creando una clase PatternList, que leerá una lista de patrones de un archivo y permitirá determinar si una posición, en un estado del juego particular, coincide con algún patrón. Debido a que trabajamos con patrones de un tamaño menor, de 3x3, se logra implementar el algoritmo de manera de poder hacer el chequeo en tiempo constante. Básicamente:
     - Se crea un arreglo de $3^9 = 19683$ booleanos, representando cada posición una posible combinación de un tablero de 3x3 y se lo inicializa en false.
     - Por cada patrón leído del archivo (dado en un formato general definido) se generan todos los posibles tableros 3x3 que coincidirán con dicho patrón, incluyendo simetrías y rotaciones. Por cada uno, se activa su posición en el arreglo, almacenando el valor true.
     - Luego, para saber si una posición de juego coincide con algún patrón, simplemente se corrobora que su posición en el arreglo tenga el valor true. Es decir, podemos hacerlo en tiempo constante O(1).
   - Modificando la clase *StateGo* para que lleve un registro de los bloques en estado de *atari* y, de esta manera, permita calcular eficientemente los movimientos de captura y escape.
   - Implementando una nueva clase *SelectResMostRobustOverLimit* que hereda de la clase *SelectRes* e implementa la selección del movimiento a realizar con la consideración antes mencionada de dar por perdidos juegos muy desfavorables.

10 Búsqueda de coeficientes óptimos

Dentro de la implementación del programa se cuenta con muchas variables que se pueden modificar e influyen en la forma que el programa progresa, como ser: la lista de patrones a utilizar, el coeficiente de exploración para el algoritmo UCT, la variable K del algoritmo RAVE, el número de visitas antes de expandir un nodo, el número de ciclos del algoritmo que se realizarán antes de cada movimiento, el número de threads a utilizar en paralelo, etc.
De manera de poder testear al programa en diferentes situaciones y así buscar los valores óptimos para cada una de estas variables, se implementaron varias opciones de linea de comando que permiten configurar cada una de ellas sin necesidad de recompilar el código (en caso de no estar presentes, se toma un valor por defecto).
Haciendo uso de estas opciones, de la aplicación "gogui-twogtp" [15] que permite correr múltiples partidas entre dos programas y de un script que se creó para generar estadísticas ("generate_stats.py"), se testeó al programa corriendo múltiples partidas modificando el valor de diferentes variables y analizando el porcentaje de partidas ganadas contra el programa *GnuGo* [14].
Coeficiente de Exploración: Como se puede ver en Fig.6, de acuerdo a las pruebas realizadas para diferentes valores del coeficiente de exploración en el algoritmo UCT, y en concordancia con los resultados presentados en [2], el valor óptimo luego de la incorporación de la mejora RAVE, resulta ser 0.
Número de simulaciones: Claramente, como se visualiza en Fig.7, con el aumento del número de simulaciones incrementará el porcentaje de partidas ganadas, porque se cuenta con un árbol cada vez más grande y más cercano al árbol min-max completo. Sin embargo, al mismo tiempo se notará un aumento en el tiempo necesario para cada toma de decisión.
En nuestro caso se lograron buenos resultados con 30000 simulaciones por mov. Ejecutando el algoritmo sobre 5 threads en paralelo, en una máquina de 4 núcleos, se requiere aproximadamente 2 segundos por movimiento.

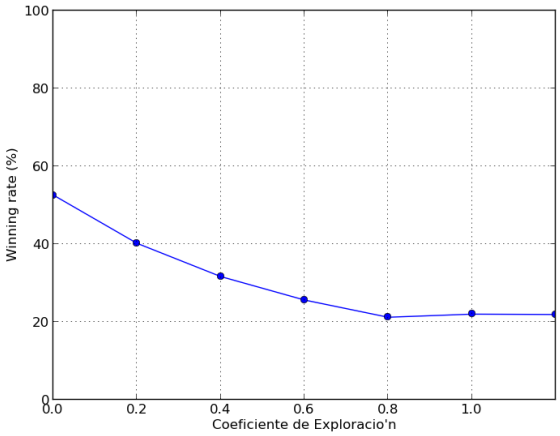

Figura 6: Análisis del coeficiente de exploración (200 partidas por caso).

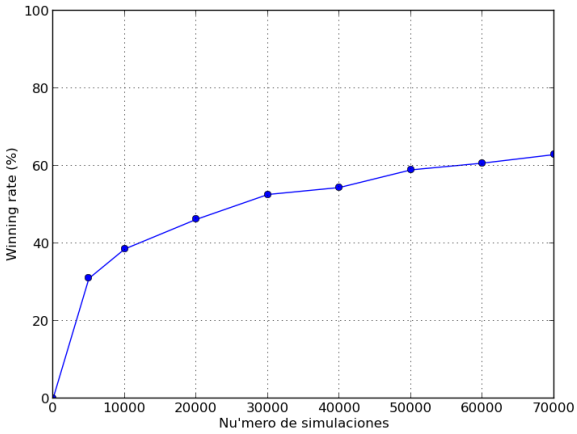

Figura 7: Análisis del número de simulaciones (300 partidas por caso).

11 Resultados

Las siguientes comparaciones se realizaron sobre un tablero de 9x9, utilizando la puntuación China (por Área) y un komi de 5 puntos.

- Resultado sobre 100 partidas entre el programa desarrollado, con simple UCT, y el mismo con la incorporación de la mejora RAVE (30000 sim. por mov.):

Programa 1	**Programa 2**	**Blanco/Negro Prog1**	**% Ganado Prog1**
UCTRave Aleatorio	UCT Aleatorio	Blanco	100%
UCTRave Aleatorio	UCT Aleatorio	Negro	100%

- Resultado sobre 100 partidas entre el programa, con simulaciones totalmente aleatorias, y el mismo con la incorporación de patrones y valores de capturas (30000 sim. por mov.):

Programa 1	**Programa 2**	**Blanco/Negro Prog1**	**% Ganado Prog1**
UCTRave con Conoc. Dominio	UCTRave Aleatorio	Blanco	93%
UCTRave con Conoc. Dominio	UCTRave Aleatorio	Negro	94%

- Resultado sobre 500 partidas contra *GnuGo-3.8 [14] level 10* (30000 sim. por mov.):

Programa	**Blanco/Negro**	**% Ganado**
UCT Aleatorio	Blanco	5%
UCT Aleatorio	Negro	3%
UCTRave Aleatorio	Blanco	14%
UCTRave Aleatorio	Negro	8%
UCTRave con Conoc. Dominio	**Blanco**	**52%**
UCTRave con Conoc. Dominio	**Negro**	**51%**

12 Conclusión

A través de la investigación respecto al juego Go y el algoritmo de Monte Carlo, este trabajo nos permitió introducirnos en un importante área de investigación dentro de la Inteligencia Artificial, incorporando un conocimiento general del estado del arte y principales lineas de investigación en la actualidad.
Se presenta como aporte el desarrollo de una librería general para el algoritmo de MCTS basada en templates, junto a múltiples ejemplos de uso.
Al mismo tiempo, haciendo uso de esta librería, se desarrolló un jugador virtual del Go. A medida que se avanzó en el desarrollo, se visualizaron grandes incrementos en la efectividad a partir de las mejoras añadidas. Al incorporar la mejora RAVE, el programa superó en todas las partidas a su versión inicial. De igual manera, el programa

superó en gran medida a su versión anterior con la incorporación de conocimiento de dominio en la etapa de simulación. Finalmente, podemos decir que se concluyó con un jugador virtual de un nivel importante, considerando el porcentaje de partidas ganadas contra el programa *GnuGo* [14], comúnmente utilizado como punto de evaluación, en un nivel muy cercano al logrado por los principales prog. del área como: *MoGo* [9], *ManGo* [7], *FueGo* [11].
Como trabajo a futuro, se presentan las siguientes alternativas:

- Incorporar conocimiento de dominio en la etapa de Selección. Por ej: "Progressive Bias - Widening" [7].
- Probar el programa con nuevos patrones e investigar la posibilidad de aprenderlos automáticamente [7].
- Mejorar el programa para que pueda regular el tiempo automáticamente de acuerdo a la etapa en la que se encuentre del juego, evaluando cuándo tiene sentido invertir más tiempo explorando y cuándo no es necesario.
- Añadir soporte para "opening books", incluyendo ciertos movimientos fundamentales al inicio de una partida.
- Mejorar la evaluación del tablero de juego de manera de poder determinar cuándo un bloque está muerto (sin posibilidades de sobrevivir si el oponente juega adecuadamente). Aspecto muy importante en la punt. japonesa.
- Analizar y optimizar el programa para tamaños de tablero distintos de 9x9, como ser 13x13 y 19x19. Estas opciones no se analizaron detenidamente en nuestro caso y pueden requerir cambios en los algoritmos de simulación y criterios de las demás etapas para lograr un jugador de un nivel aceptable.
- Implementar la etapa de simulación utilizando GPGPU. De esta manera se podrían realizar muchísimas más simulaciones en igual tiempo y definitivamente la efectividad del programa aumentaría en gran medida (Fig.7).

Referencias

[1] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/grand-challenge.pdf

[2] Sylvain Gelly and David Silver. 2011. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. Artif. Intell. 175, 11 (July 2011), 1856-1875. http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/mcrave.pdf

[3] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. https://dke.maastrichtuniversity.nl/m.winands/documents/multithreadedMCTS2.pdf

[4] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. http://www.sztaki.hu/~szcsaba/papers/ecml06.pdf

[5] Guillaume Maurice Jean-Bernard Chaslot. Monte-Carlo Tree Search. https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf

[6] Guillaume Chaslot, Louis Chatriot, C. Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud. Combining expert, offline, transient and online knowledge in Monte-Carlo exploration. https://www.lri.fr/~teytaud/eg.pdf

[7] Guillaume Chaslot, Mark H.M. Winands, H. Jaap van den Herik, Jos W.H.M. Uiterwijk, Bruno Bouzy. Progressive Strategies for Monte-Carlo tree search. https://gnunet.org/sites/default/files/NMNC%20-%20Progressive%20strategies%20for%20MCTS.pdf

[8] Sylvain Gelly, Yizao Wang, Modifications of UCT and sequence-like simulations for Monte-Carlo Go. http://dept.stat.lsa.umich.edu/~yizwang/publications/wang07modifications.pdf

[9] Sylvain Gelly, Yizao Wang, Rémi Munos, Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. https://hal.inria.fr/inria-00117266/PDF/RR-6062.pdf

[10] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. https://hal.archives-ouvertes.fr/file/index/docid/369786/filename/TCIAIG-2008-0010_Accepted_.pdf

[11] Markus Enzenberger, Martin Muller, Broderick Arneson and Richard Segal. FUEGO - An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search. http://webdocs.cs.ualberta.ca/~mmueller/ps/fuego-TCIAIG.pdf

[12] Beatriz Nasarre Embid. Método de Monte-Carlo Tree Search (MCTS) para resolver problemas de alta complejidad: Jugador virtual para el juego del Go. http://zaguan.unizar.es/record/8010/files/TAZ-PFC-2012-393.pdf

[13] Georgy Adelson-Velsky, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

[14] GNU Go. https://www.gnu.org/software/gnugo/

[15] GoGui. http://gogui.sourceforge.net/

[16] Go Text Protocol. http://www.gnu.org/software/gnugo/gnugo_19.html

[17] GNU General Public License http://www.gnu.org/copyleft/gpl.html