Estrategia de resolución Minimax para Juegos de Suma Cero e Implementación para el juego *BlobWars*

Kevin J. Hanna[1] y Pablo A. Costesich[2]

Alumnos de Ingeniería Informática
Instituto Tecnológico de Buenos Aires
Av. Madero 399, C.A.B.A., Argentina
[1]*khanna@alu.itba.edu.ar*
[2]*pcostesi@ieee.org*

11 de noviembre de 2013

Resumen

Se presenta una introducción a la Teoría de Juegos y Juegos de Suma Cero. Estos son juegos de dos jugadores con turnos y un balance entre sus ganancias y pérdidas. La estrategia de resolución analizada en este informe es Minimax, un algoritmo de búsqueda en profundidad limitada, junto con la heurística de Poda Alfa-Beta. Se detalla la implementación y resultados prácticos.

1. Introducción

El Trabajo Práctico Especial de la cátedra de Estructuras de Datos y Algoritmos para el segundo cuatrimestre de 2012 propone la resolución del juego *BlobWars* mediante el algoritmo Minimax, con y sin Poda Alfa-Beta, en el lenguaje de programación Java. La implementación del mismo debe contar con un modo visual y de lectura de tableros por archivo.

1.1. BlobWars

BlobWars es un juego de dos jugadores por turnos en un tablero de 8x8. El objetivo es lograr la mayor cantidad de piezas sobre el tablero cuando alguno de los jugadores no pueda realizar nuevos movimientos.

Los posibles movimientos de una mancha (*blob*) son siempre hacia casilleros vacíos y cumplen estas reglas:

- Los movimientos a distancia 1 mantienen la mancha de origen y generan una nueva del mismo color en el casillero de destino.
- Los movimientos a distancia 2 desplazan la mancha al casillero de destino.

- No pueden realizarse movimientos que no sean de distancia 1 o 2.

Definición El conjunto $Manchas$ es equivalente a $\{-1, 0, 1\}$.

Definición Un tablero es una matriz de $Manchas^{8x8}$.

Definición Un punto es un par ordenado $\langle x, y\rangle \in Point$, donde $Point = Side \times Side$ y $Side = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Definición Se llama $Movements$ al siguiente conjunto:

$$Movements = Point \times Point \tag{1}$$

Definición Un movimiento es un par ordenado $\langle inicio, destino\rangle \in Movements$.
Notación: $inicio \rightarrow destino$.

Definición Un movimiento válido para un jugador p es un movimiento tal que el casillero de destino d se encuentra libre (su valor es 0) y el de inicio s pertenece al jugador p (su valor es p).
Notación: $v_{s \rightarrow d}(p)$

Definición Un casillero es el valor del tablero en un punto p.

Definición El casillero vacío es equivalente al valor 0 en el tablero para el punto p.

Definición Una mancha es uno de los tres posibles estados $\in Manchas$ de una celda en un punto del tablero:

-1 *Humano*: representado por la letra *H*.

0 *Vacío*: representado por el espacio en blanco.

1 *Computadora*: representado por la letra *C*.

Definición Se define distancia como:

$$distancia(a, b) = \text{máx}(|a[x] - b[x]|, |a[y] - b[y]|) \tag{2}$$

Donde a y b son los puntos en cuestión (tuplas de dos componentes $\mathbb{N}_0$), y $a[x]$ significa la componente x de a.

Adicionalmente, al final de cada movimiento la mancha en el destino *infecta* a las vecinas según estas reglas:

- Si el casillero no se encuentra ocupado entonces se deja libre (no se altera).
- Si el casillero se encuentra ocupado, la mancha presente en éste cambia al color de la mancha que infecta.

Definición Aplicar un movimiento m para un jugador p a un tablero b es aplicar las reglas de movimiento seguido de *infectar* el casillero de destino. Se obtiene como resultado un tablero b'

Definición Un *Tablero Terminal* es un tablero tal que para algún jugador no existen movimientos válidos para ninguna de sus manchas.

$$terminal(b) \Leftrightarrow \nexists m \in v_{s \to d}(p) \forall s, d \in Movements \tag{3}$$

Donde b es el tablero, m un movimiento válido, p un jugador y s, d un movimiento.

El juego termina cuando el tablero se completa o cuando el siguiente jugador no puede realizar un movimiento válido. Vale destacar que la primera condición implica la segunda.

Definición Un turno es aplicar un movimiento válido m para un jugador p a un tablero b y retornar el tablero resultante b':

$$b' = turno(b, p, m) \tag{4}$$

Definición Un *ply* es un grupo de dos turnos que resulta de la aplicación recursiva de éstos para dos movimientos de jugadores alternados m_p y $m'_{p'}$. En otras palabras:

$$ply(b, m_p, m_{p'}) = turno(turno(b, p, m_p), p', m_{p'}) \tag{5}$$

Definición Una partida es una serie finita G de n *plies* tal que aplicar el último movimiento $(m^n_{s \to p})$ al tablero b^{n-1} resulta en un tablero terminal b^n; siendo b^n el único tablero terminal en G.

2. Teoría de Juegos

La Teoría de Juegos es un campo de la matemática que estudia mediante modelos sistemas definidos por reglas de acción con incentivos (juegos), y proporciona estrategias para llevar a cabo procesos de decisión. Es un campo de estudio estrechamente ligado a las ciencias del comportamiento y las ciencias que las utilizan, como ser: economía, biología, psicología, antropología y ciencias políticas. [BBS, p. 1]

El análisis de estos modelos implica una característica diferencial con respecto a otros campos como la Teoría de la Decisión en cuanto a que los actores en la Teoría de Juegos no tienen costos e incentivos prefijados, sino que varían con las acciones de sus oponentes. También asume que cada jugador elige según un criterio de *fitness* o ventaja sobre su situación y no mediante un set de reglas independiente de la misma.

Definición *Fitness* es una función de probabilidad que toma un estado o conjunto de estímulos y devuelve una valoración escalar.

Mediante el ingreso de estímulos a una constelación de entradas sensoriales, el actor racional genera una distribución estadística de probabilidad sobre distintos resultados con un *fitness* asociado. [BBS, p. 4]

Los actores racionales escogen el resultado con mejor *fitness* para un entorno determinado. Por lo tanto, si un actor es presentado con una opción A de menor *fitness* que B escogerá B, y si es presentado con dicho B de menor *fitness* que C

entonces escogerá C por sobre A y B. Esto es llamado *Consistencia de Elección.* [BBS, p. 4]

Este modelo se llama *BCP* (*beliefs, preferences, and constraints* [BBS, p. 3]), e implica que distintas decisiones afectan a los resultados de las valuaciones de *fitness* con distinto grado de probabilidad. La Teoría de Juegos amplía el modelo para incorporar varios jugadores, y llama Estrategias a las *BCP*. Los jugadores cuentan con el conocimiento de las reglas del juego, la naturaleza de sus contrincantes y estrategias disponibles. Para cada combinación de decisiones, para cada jugador, el juego le asigna una distribución de *individual payoffs* (ganancias individuales). Con esto, la Teoría de Juegos logra predecir el comportamiento de cada jugador (al asumir que maximizará sus ganancias, al ser un actor racional). [BBS, p. 8]

Cabe destacar que el modelo no es aplicable a todo tipo de modelo válido, ya que en la práctica puede ocurrir que los actores no sean enteramente racionales, desconozcan parte de la información en un estado determinado o no sean consistentes en sus decisiones (por ejemplo, un animal que se vuelve adicto a un alimento determinado). [BBS, p. 9 - 11,]

El problema más grave es que los actores racionales no respeten el Equilibrio de Nash. [RAM]

2.1. Equilibrio de Nash

El Equilibrio de Nash es un concepto de solución para un juego no cooperativo de dos o más jugadores, en donde todos ellos tienen conocimiento de las estrategias de equilibrio de los otros.

Si cada jugador ha elegido una estrategia y ningún jugador puede beneficiarse de un cambio de la misma cuando sus oponentes mantienen las de ellos, entonces se afirma que dicho conjunto de estrategias y sus ganancias individuales constituye un Equilibrio de Nash. [OSB]

Si un jugador A tiene una estrategia dominante S_A entonces existe un Equilibrio de Nash en donde A juega S_A. En el caso de ser dos jugadores A y B, existe un Equilibrio en donde A juega S_A y B juega la mejor estrategia en respuesta a S_A (puede no ser única). Si S_A es una estrategia estrictamente dominante, A juega S_A en todo Equilibrio de Nash. Si ambos A y B tienen estrategias estricamente dominantes, existe un único Equilibrio de Nash en donde ambos jugadores juegan sus estrategias dominantes respectivas.

2.2. Juegos de Suma Cero

Los juegos de suma cero responden a la premisa básica de que las ganancias de un jugador se equiparan a la pérdida conjunta de los otros. Asume que existe un Equilibrio de Nash para un el mismo.

Definición Una estrategia mixta es un tipo de estrategia donde un jugador escoge una columna de la matriz de ganancias del juego, mientras que el adversario escoge una fila. El conjunto de valores de dicha celda corresponde a las ganancias individuales para cada jugador.

En el caso de los juegos de dos jugadores, un Equilibrio de Nash y Minimax devuelven la misma solución, donde constituye una estrategia mixta.

La resolución de los mismos puede llevarse a cabo mediante programación lineal, en donde dada una matriz de ganancias M, tal que el elemento $M_{i,j}$ es la ganancia individual cuando el jugador que minimiza escoge la estrategia en la fila i y el que maximiza escoge la estrategia en la columna j, se debe encontrar un vector u que cumpla:

Minimizar:

$$\sum_i u_i \tag{6}$$

Tal que:

1. $u \geq 0$
2. $Mu \geq 1$

[MIC]

3. Estructuras de Datos

Para representar el árbol de juego se utilizó el propio stack de la máquina, y se aprovechó que la generación de tableros pudo ser realizada en un iterador. Esto supuso un ahorro en la implementación de estructuras caras como HashMaps y Listas Enlazadas. El ahorro no sólo fue espacial, sino que también temporal ya que no se necesitó crear gran cantidad de estructuras por cada nodo.

El tablero se representó mediante un array de objetos (referencias) a Player. Como el tablero tiene una altura fija, convertir los índices a posiciones y vice versa resultó sencillo y barato. Este tipo de representación se decidió por una cuestión de simplicidad (ahorrar la creación de Arrays anidados) y por la indirección que genera desreferenciar un objeto.

Point se implementó al principio como una clase con un patrón constructor que almacenaba las instancias de puntos anteriormente entregados para reducir el gasto de memoria. Al eliminar el uso de las listas como almacenamiento de tableros hijos, el uso de puntos en un momento dado en memoria se redujo y se volvió un simple contenedor de enteros.

Movement se implementó de forma tal en que ahorrara escribir $Pair\ \langle x, y \rangle$. Es otro simple contenedor con un método que retorna la distancia entre los puntos del par.

Pair se implementó a modo de una tupla de dos objetos.

4. Algoritmos

Se implementaron dos familias de algoritmos, cada una con dos sub-categorías de heurística:

- TBID{ABP, L}Minimax: familia de algoritmos basados en tiempo, con *Iterative Deepening*.
- {ABP, L}Minimax: familia de algoritmos basados en nivel.

Las subcategorías de heurística corresponden a:

- ABP: *Alpha Beta Pruning*, o Poda Alfa Beta. Elimina estados de búsqueda si la Ventana Alfa Beta se cierra.
- L: *Level*, Naïve Minimax basado en nivel (sin poda).

Si bien *ABP* es similar a *L* en su implementación, algunos detalles del lenguaje evitaron la reutilización directa del código.

4.1. Minimax (Naïve)

Minimax se basa en el concepto de Equilibrio de Nash y Juegos de Suma Cero, donde *max*imizar las ganancias significa *min*imizar las pérdidas. Asume que cada jugador conoce las reglas del juego, las estrategias del oponente y que siempre actuará con la mejor disponible (ya que se encuentra en equilibrio).

La versión teórica de minimax no está determinada por nivel y evalúa hasta llegar a un tablero terminal. Sin embargo, el problema de generar tableros entra en la categoría NP, por lo que es imprático implementarlo de este modo.

El siguiente es un ejemplo (funcional) en Python:

```python
def minimax(board, maximize=True, is_computer=True):
    """ Naïve minimax solver. """
    # set the worst score possible for this player.
    score = -board.max_score if maximize else board.max_score
    best_move = None
    for child, move in board.children(is_computer=is_computer):
        local, _ = minimax(child, level - 1, not maximize, not is_computer)
        # update the score when a better local solution is found
        if (maximize and score < local) or (not maximize and score > local):
            score = local
            best_move = move
    # return the score when a terminal board has been reached
    if best_move is None:
        return board.score, None
    return score, best_move
```

Nota: el ejemplo, al igual que los subsiguientes, devuelve un *score* con un movimiento.

4.2. Minimax (Level)

Minimax Level (por Niveles) agrega una condición de corte al llegar a un nivel máximo dentro del árbol de llamados recursivo.

```
def minimax(board, level, maximize=True, is_computer=True):
    """ Naïve minimax solver, using level cut-off. """
    if level == 0:
        return board.score, None
    # set the worst score possible for this player.
    score = -board.max_score if maximize else board.max_score
    best_move = None
    for child, move in board.children(is_computer=is_computer):
        local, _ = minimax(child, level - 1, not maximize, not is_computer)
        # update the score when a better local solution is found
        if (maximize and score < local) or (not maximize and score > local):
            score = local
            best_move = move
    if best_move is None:
        return board.score, None
    return score, best_move
```

4.3. ABPMinimax (Minimax con Poda Alfa-Beta)

La poda Alfa Beta es una modificación al algoritmo original de Minimax (pero se presenta en el informe como una versión de LMinimax modificada), de forma tal donde existe una *Ventana Alfa Beta* de operación. Dicha ventana está formada por dos variables α y β, que corresponden respectivamente con el jugador que maximiza y el que minimiza.

La poda tiene la característica de devolver una solución con el mismo costo que Minimax, con la diferencia de no tener que explorar soluciones innecesarias.

El primer llamado se realiza bajo los valores normales de Minimax (el *score* mínimo posible para el jugador que maximiza es asignado a α y el *score* máximo posible para el jugador que minimiza es asignado a β).

Para la etapa de maximización, ABPMinimax explora las posibles soluciones y actualiza el valor de α de forma tal que quede con el máximo *score* hasta el momento. La condición de corte (llamada *Beta Cut-Off*) añadida es que si $score \geq \beta$ deja de explorar soluciones, ya que el valor excede la ventana $\alpha \leq score \leq \beta$.

Para la etapa de minimización, actualiza el valor de β con el mínimo *score* hasta el momento. La condición de corte (*Alpha Cut-Off*) es: $score \leq \alpha$.

```
def alphabeta(board, level, alpha, beta, maximize=True, is_computer=True):
    """ Minimax with Alpha-Beta Pruning. """
    if level == 0:
        return board.score, None
    # we base the score on the previous runs (alpha-beta window)
    score = alpha if maximize else beta
    best_move = None
    for child, move in board.children(is_computer):
        # find a new local solution
        local, _ = alphabeta(child, level - 1, alpha, beta,
                not maximize, not is_computer)
        # Update the score when a better local solution is found
        if (maximize and score < local) or (not maximize and score > local):
            score = local
            best_move = move
        # Update alpha or beta (this narrows the alpha-beta window)
        if maximize:
            alpha = max(score, alpha)
        else:
            beta = min(score, beta)
        # If we fall outside the alpha-beta window, return.
        if (maximize and score >= beta) or (not maximize and score <= alpha):
            break
    return score, best_move
```

4.4. TBID* (Time Bound Iterative Deepening)

Iterative Deepening es una técnica donde se ejecuta varias tiradas de un algoritmo con un aumento progresivo en algún parámetro (profundidad, en el caso de Minimax) mientras haya tiempo disponible. Se almacena sólo la solución de la ejecución completa con mayor profundidad.

Dado que el tiempo de ejecución de Minimax varía según el tablero y la profundidad, *Iterative Deepening* provee un marco estable temporal a expensas de estados explorados.

5. Problemas encontrados y decisiones tomadas

5.1. Uso eficiente de memoria

Inicialmente cuando se desarrollaron las clases del juego, se notó que consumían memoria demás. Eso se vió demostrado ya que el algoritmo propuesto para minimax no podía llegar hasta el nivel 5. A continuación se detallan los puntos críticos en donde se tuvo que repensar la implementación.

5.2. Clase Board

En un principio, la forma utilizada para representar el board fue utilizando un array de $Tile$, del cual heredaban $EmptyTile$ y $BlobTile$. Es decir, que la cantidad de referencias de cada tablero era constante. Por cada tablero se guardaban 64 $(8 * 8)$ referencias a algún $Tile$. A la vez, cada $BlobTile$ contenía una referencia a un $Player$.

Para iterar facilmente por los tiles de un Player y obtener sus jugadas posibles se contó con un $HashMap < Player, ArrayList < Point >>$, y un método para acceder a esos puntos en donde se encontraban los tiles de cada jugador. Lo cual se descartó para hacer uso más eficiente de la memoria.

Al eliminar el mapa que referenciaba jugador con sus tiles, se decidió implementar el Board cómo un array primitivo de Player, y un método para convertir de Point al indice de ese array. En este caso se hizo un cambio que consume menos memoria al haber menos referencias y objetos en tiempo de ejecucón, aunque es cierto que esa solución hace que haya que calcular más en tiempo de ejecución.

El cambio más radical en cuanto a cantidad de memoria consumida, se logró implementando un iterador de Board para ser utilizado al generar los tableros con las jugadas posibles a partir de cierto Board.

5.3. Clase Point

La implementación de esta clase fue variando a lo largo del desarrollo del proyecto. En una primera instancia se utilizo la clase $Point$ que provee Java. Al ver que estabamos desperdiciando memoria utilizando muchos objetos $Point$ con los mismos valores (x, y), se decidió implementar una clase $Point$ propia. La clase contaba con un método de clase $getInstance(int, int)$, si el punto ya estaba creado se devolvía una referencia e él. Sino, se creaba un punto nuevo. La implementación de $getInstance()$ se descartó después de haber implementado un iterador en la clase Board, ya que los $Point$ dejaron de ser tantos en memoria.

Esta clase, ya no sería de de dos coordenads *double*, sino que de *int*. Los beneficios fueron altos ya que además de ocupar menos memoria, no se debía castear de *double* a *int* para acceder al array.

6. Comparaciones y pruebas

Se probó con dos tableros aleatorios distintos el tiempo de ejecución para las cuatro posibles combinaciones de *solvers*.

6.1. ABPMinimax

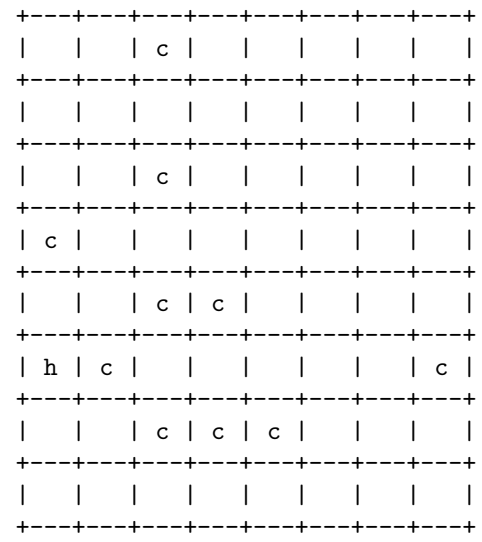
El tablero 1:

```
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| c |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c | c |   |   |   |   |
+---+---+---+---+---+---+---+---+
| h | c |   |   |   |   |   | c |
+---+---+---+---+---+---+---+---+
|   |   | c | c | c |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

Tablero	Jugador	Profundidad	Tiempo de Ejecución	Estados Explorados
1	2	1	6ms	147
1	2	2	17ms	403
1	2	3	223ms	53298
1	2	4	878ms	119559
1	2	5	7037ms	9276453
1	2	6	31707ms	19059529

6.2. Level Minimax

El tablero 1:



Tablero	Jugador	Profundidad	Tiempo de Ejecución	Estados Explorados
1	2	1	14ms	147
1	2	2	41ms	1405
1	2	3	736ms	152930
1	2	4	8474ms	5495678

Nota: Level Minimax tiene un bug por el cual el nivel par no es correctamente reportado (min no suma correctamente la variable level). Los números de nivel han sido ajustados.

6.3. TBIDABPMinimax

El tablero 1:

```
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| c |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c | c |   |   |   |   |
+---+---+---+---+---+---+---+---+
| h | c |   |   |   |   |   | c |
+---+---+---+---+---+---+---+---+
|   |   | c | c | c |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

Tablero	Jugador	Profundidad	Tiempo de Ejecución	Estados Explorados
1	2	3	1001ms	53848
1	2	4	2000ms	173407
1	2	4	4001ms	173407
1	2	4	8002ms	173407
1	2	5	16000ms	9449860
1	2	5	32001ms	9449860

Nota: Las implementaciones de *TBID* no tienen en cuenta los estados explorados en las iteraciones que no fueron completadas, ya que no afectan al resultado del algoritmo.

6.4. TBIDLMinimax

El tablero 1:

```
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| c |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | c | c |   |   |   |   |
+---+---+---+---+---+---+---+---+
| h | c |   |   |   |   |   | c |
+---+---+---+---+---+---+---+---+
|   |   | c | c | c |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

Tablero	Jugador	Profundidad	Tiempo de Ejecución	Estados Explorados
1	2	3	1001ms	154482
1	2	3	2000ms	154482
1	2	3	4000ms	154482
1	2	3	8000ms	154482
1	2	3	16001ms	154482

Nota: Las implementaciones de *TBID* no tienen en cuenta los estados explorados en las iteraciones que no fueron completadas, ya que no afectan al resultado del algoritmo.

7. Implementación

7.1. Chain

Dado el uso extensivo de iteradores en el código, se sucitó la necesidad de una forma de concatenarlos. `Chain` se planteó como un `Iterable<E>` para un conjunto de `Iterable<E>`, que retornara un `Iterator<E>` con la concatenación de los iteradores de la lista de iterables.

Está basada en la función `chain()` de `itertools` en la librería estándar de Python.

7.2. Paquete `optparse`

El parseo de la línea de comandos en Java resultó difícil de encarar mediante el uso crudo de `String[] args`. Se decidió crear un paquete con una clase `OptionParser` que fuera un contenedor de objetos `Option`, quienes fueron diseñados para realizar el parseo localmente.

Dos subclases retornan distintos comportamientos para distintos tipos de opciones: `FlagOption` parsea opciones sin aridad, mientras que `ValuedOption` parsea opciones de ariedad arbitraria.

8. Conclusiones

De las observaciones de los datos se puede destacar que:

- La heurística de Poda Alfa-Beta es efectiva y reduce el tiempo de ejecución y espacio en memoria (este último no medido en *benchmarks*) necesarios, mientras que la solución parcial sigue siendo equivalente.
- La familia de algoritmos *TBID* logra un mejor desempeño para ciertos tableros (dependiendo de la cantidad de movimientos válidos para ambos jugadores), ya que la cantidad de movimientos posibles no es constante. Puede explorar más niveles en partidas cercanas a ser finales.
- El uso de iteradores y el uso del propio stack de la JVM mejora sustancialmente el desempeño y vale por el tiempo invertido en utilizar estas técnicas. Se desaconseja el uso de listas y mapas para código que es accedido regularmente (*hotspots*).

9. Anexo

9.1. Ejemplo de implementación del Solver en Python

Para cruzar datos con el código en Java y para generar los tableros al azar se implementó un script de Python que validara los movimientos. El mismo se encuentra en: https://gist.github.com/4018933

9.2. Implementación alternativa del núcleo del Solver

Se consideró una tercera implementación de *Solvers*, basados en *TBID* y particionamiento horizontal, en el cual levantara tantos *workers* como núcleos disponibles en el equipo, cada uno con un nivel distinto. Cuando un *terminara*, éste recibiría el siguiente nivel del grupo.

En teoría, cada *worker* se ejecutaría en un núcleo distinto con distintos niveles. Por lo tanto, en una máquina de 8 núcleos, podría explorar desde el nivel 1 a 8 y lograr una profundidad igual o mayor que *TBID* para el mismo tiempo, ya que no arrastraría el tiempo de ejecución de niveles anteriores (para los primeros 8 niveles).

Esta solución se descartó dado que la complejidad inherente del mismo, si bien no muy distinta a la de *TBID*, no traía mejoras considerables fuera de los primeros pasos.

Referencias

[BBS] *A framework for the unification of the behavioral sciences*, Herbert Gintis, Behavioral and Brain Sciences (2007) 30:1-61

[RAM] Chen, H.-C., Friedman, J. W. & Thisse, J.-F. (1997) Boundedly rational Nash equilibrium: A probablistic choice approach. Games and Economic Behavior 18:32– 54.

[OSB] Osborne, Martin J., and Ariel Rubinstein. A Course in Game Theory. Cambridge, MA: MIT, 1994. Print.

[MIC] Bowles, Samuel (2004). Microeconomics: Behavior, Institutions, and Evolution. Princeton University Press. pp. 33–36. ISBN 0-691-09163-3.