SNRPA Concurrente Aplicado al Problema TSPTW

Sergio Andrés Rodríguez Torres
Estudiante máster en informatica
Escuela colombiana de ingeniería Julio Garavito
Bogotá, Colombia
sergio.rodriguez-tor@mail.escuelaing.edu.co

***Resumen*—A lo largo de este artículo se abordará sobre el problema del agente viajero con ventanas de tiempo usando una versión concurrente con un modelo de actores de búsquedas recursivas de Monte-Carlo con adaptación de políticas.**

***Index Terms*—TSPTW, NRPA, algoritmia, concurrencia**

I. Introducción

Este articulo busca explorar un solución al problema del agente viajero con ventanas de tiempo (TSPTW), uniendo el conocimiento de varios artículos relacionados al refinamiento de estrategias basadas en búsquedas de aleatorias de Monte-Carlo se explican detalles del algoritmo y optimizaciones hechas al aplicarlo al problema.

El artículo se estructura de la siguiente forma. Primero se dará contexto sobre el problema abarcando el problema TSPTW, luego sobre el algoritmo de búsquedas recursivas de Monte-Carlo con adaptación de políticas estable (SNRPA) propuesto por Tristan Cazenave, luego tocaremos algunos detalles de la implementación de SNRPA para TSPW para optimizar al máximo la solución incluido el modelo de actores para manejar la concurrencia del algoritmo. Por último, se hablará del trabajo futuro y conclusiones.

II. Contexto

II-A. TSPTW

El problema del agente viajero TSP dadas una lista de ciudades (nodos) y distancias entre ellas, cual es la ruta más corta posible que visita cada ciudad una vez y finaliza en la ciudad de origen. Este problema ha sido probado ser NP-hard, pero es un problema vital para la planificación y logísticas de productos en mundo real. Por eso a pesar de su dificultad es importante estudiar algoritmos y heurísticas que permitan encontrar mejores soluciones el menor tiempo posible.

II-B. Soluciones propuesta

II-B1. NRPA: Tristan Cazenav propuso una forma de adaptar NRPA a TSPTW, también propuso varios artículos que refinan la solución como GNRPA o SNRPA.

II-B2. Redes que se expanden en el tiempo: Natashia Boland, Mike Hewitt y Vu Duc Minh propusieron una red que usa nodos que se forman de las ciudades y el momento, por lo que la red crece a medida que se hacen movimientos.

II-B3. Heurística: López Ibáñez, Christian Blumb implementaron un algoritmo hibrido que combina una estrategia de colonia de hormigas mezclada con beam-search usando muestreos estocásticos. esta solución es especialmente interesante para optimizar el costo del viaje.

III. NRPA

III-A. ¿Que es?

La exploración del árbol de búsqueda de Monte-Carlo (MCTS) ha sido aplicado muchos juegos y problemas [2], este algoritmo se popularizó tras obtener resultados notables en la solución de juegos complejos como Go y ajedrez los cuales tiene la particularidad de tener un espacio de búsqueda muy grande en términos de la cantidad de estados que pueden derivar de un movimiento en el juego y por ende es difícil obtener una secuencia de estados que lleven a la solución del problema [3]. Derivado de este algoritmo surge la búsqueda recursiva de Monte-Carlo (NMCS) para la solución de problemas de optimización, esta sesga la elección aleatoria de un movimiento en base a una distribución probabilística que llaman política. Más recientemente se ha modificado este algoritmo para actualizar su política a través de una árbol recursivo usando una estrategia similar a Backpropagation, esta versión se conoce como Nested Rollout Policy Adaptation (NRPA) [4]

III-B. ¿Como funciona en NRPA?

NRPA crea un árbol de recursión el cual es recorrido en profundidad desde la izquierda, el funcionamiento del *algoritmo 1* se muestra gráficamente en la *figura 1* cada nivel *l* hace una cantidad fija de llamados recursivos al siguiente nivel, llamaremos a esa variable *N* por lo que el árbol resultante va a se un árbol N-ario, esto se hace en el for de la línea 7.

En el nivel inferior (0-1 dependiendo de la implementación) es donde se va a calcular una posible solución que llamaremos *rollout*, esta se generará de forma aleatoria siguiendo la distribución descrita por la política. En el caso del TSPTW es un tour o permutación de todos los nodos del grafo que empiecen en el depósito y termine en el depósito esto se hace en las líneas 2-3, el detalle de cómo realizar dicho cálculo se abarcará después.

Luego con el *rollout* obtenido se toma el mejor maximizando el puntaje de este, el mejor *rollout* conocido hasta el momento en este nivel junto con listado de posibles movimientos que se podían realizar en cada paso son almacenados (lianas

10-13) para luego ser usados en el proceso de adaptación de la política (línea 14)

```
1 NRPA(level, policy)
2  if level.id == 0 then
3   return playout(policy)
4  else
5   level.bestRollout.score ← −∞
6   level.policy ← policy
7   for N iterations do
8     nextLevel ← DataPerLevel[level.id − 1]
9     bestRollout ← NRPA(nextLevel, level.policy)
10    if bestRollout.score ≥ level.bestRollout.score
          then
11      level.bestRollout ← bestRollout
12      level.movesPerStep ← nextLevel.movesPerStep
13    end if
14    level.AdaptPolicy()
15  end for
16  return level.bestRollout
17 end if
```

Algoritmo 1. El algoritmo NRPA

Como se mencionó posterior mente la política representa una distribución que describe las probabilidades de realizar un movimiento, en el mundo del TSPTW un movimiento es hacer un viaje de un nodo *u* a otro nodo *v*, entonces se representa la política como una matriz de valores de coma flotante de tamaño *V*x*V* sindo *V* el número de nodos del problema, de forma tal que la probabilidad de hacer un viaje de *u* a *v* sea descrita por el valor *policy[u][v]*.

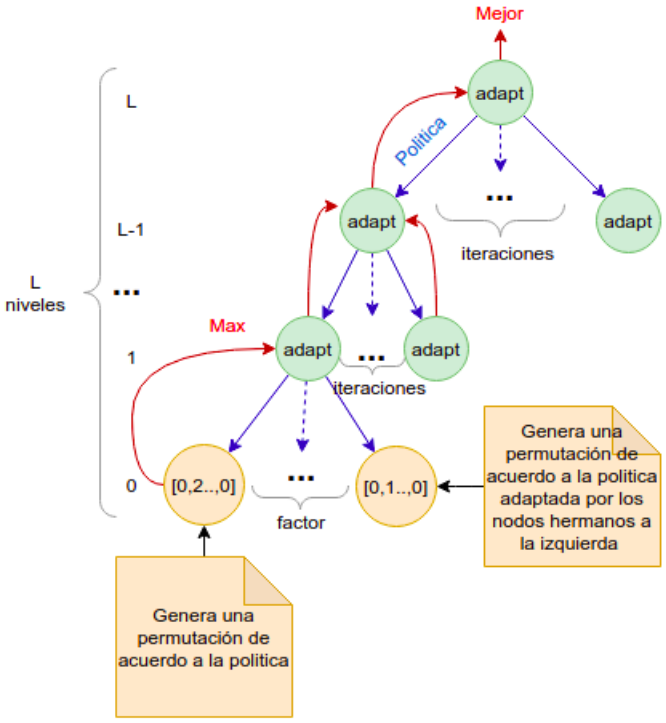

Figura 1. Representacion grafica del arbol recursivo de NRPA

El *algoritmo 2* implementa el proceso de adaptación de una política, este busca alterar la política de forma tal en la que se aumente la probabilidad elegir el actual mejor *rollout* del nivel. Esto se logra interando por todos los movimientos en la secuencia del *rollout*, en el mundo del TSPTW la secuencia es el tour, en cada paso se tomó la decisión de hacer un movimientos entre una lista de posibles movimientos que se podían realizar en ese estado, por ejemplo si ignoramos las condiciones de la ventanas de tiempo con el fin de simplificar el ejemplo, en el primer movimiento $u = 0$ (el depósito) era posible realizar un viaje desde *u* a cualquier nodo, se tomó una decisión y se realizó un movimiento de *u* a *v* siendo *v* el nodo elegido en el *rollout*, por lo que incrementaremos *policy[u][v]* en α (líneas 5-8).

```
1 (l Level) AdaptPolicy()
2   rollout ← l.bestRollout
3   polCpy ← policy
4   u ← 0 //starts at the depot
5   for step in l.movesPerStep do
6     v ← rollout.tour[step]
7     moves ← l.movesPerStep[step]
8     l.policy[u][v]  ← l.policy[u][v] + α
9     z ← 0.0
10    for m in moves do
11      k ← moves[m]
12      z ← z + e^polCpy[u][k]
13    end for
14    for m in moves do
15      k ← moves[m]
16      l.policy[u][k] −= α * e^polCpy[u][k] / z
17    end for
18    u ← v
19  end for
```

Algoritmo 2. El algoritmo de adptación para una politica

Luego debemos renormalizar la distribución y penalizar la probabilidad de hacer un movimiento *u* a *k* siendo k un nodo dentro de los posibles movimientos que se podían hacer en ese estado, el nodo *v* no será penalizado solo normalizado ya que previamente incrementaremos *policy[u][v]* en α (lineas 10-17).

```
1(r Rollout) playout(policy)
2   u ← 0 //starts at the depot
3   for step from 0 to V do
4     moves ← r.calculateLegalMoves(u, step)
5     v ← r.pickMove(u, step, policy, moves)
6     r.move(u, v)
7     u ← v
8   end for
9   r.move(u, 0) // go back to the depot
10  r.Score = −(offset∗r.Violations + r.cost)
11  return r
```

Algoritmo 3. Generación de un tour

El *algoritmo 3* genera un tour iterando *V* veces calculando los movimientos que son legales para el estado actual con el nodo actual, luego de esos movimientos elegir de acuerdo a la distribución de la politica, realizamos el movimiento y avanzamos. Por último, debemos volver al depósito y calcular el puntaje del tour elegido.

El costo del tour se definirá de la siguiente forma:

$$costo(T) = \sum_{k=0}^{V} distancia(T_k, T_{k+1})$$

suponga una función $\phi(T)$ retorna el número de violaciones de las ventanas de tiempo del tour T entonces el puntaje del tour es:

$$puntaje(T) = costo(T) + 10^6 * \phi(T)$$

El *algoritmo 3* calcula los movimientos permitidos que pueden hacer desde *u* en el paso *step*, las líneas 2-8 buscan viajes que directamente incumplan una ventana de tiempo, la estrategia es tomar esos viajes primero para deshacerse lo más pronto posible de todos los nodos que no van a resultar en un recorrido exitoso además dentro de los movimientos que se podían hacer en este estado no se tienen en cuenta otros movimientos que si son válidos, por lo que no verán afectados negativamente por no ser elegidos. Además, por la desigualdad triangular no existe un camino $u \rightarrow k \rightarrow v$ más rápido que $u \rightarrow v$ que no viole la ventana de tiempo.

Las líneas 9-26 buscan movimientos validos que no generen violaciones futuras de las ventanas de tiempo, es decir, si $u \rightarrow k$ es válido que tomar $u \rightarrow v$ no haga imposible ir de $v \rightarrow k$ en el siguiente paso

Por último, las líneas 27-32, son en caso de que todos los viajes $u \rightarrow v$ generan situaciones futuras que violan las ventanas de tiempo, entonces todos los nodos *v* son posibles movimientos.

```
1(r Rollout) calculateLegalMoves(u, step)
2 for v from 1 to V do
3   if not visited[v] then
4     if r.Makespan+Distances[u][v] > WindowEnd[v]
          then
5       legalMovesPerStep[step].append(v)
6     end if
7   end if
8 end for
9 if there are no moves in legalMovesPerStep[step]
      then
10  for v from 1 to V do
11    if not visited[v] then
12      impossibleMove ← false
13      for k from 1 to V if there's no
            impossibleMove do
14        if not visited[v] then
15          if r.Makespan <= WindowEnd[k] && r.
                Makespan+Distances[u][k] <=
                WindowEnd[k] AND
16            (r.Makespan+Distances[u][v] >
                  WindowEnd[k] OR WindowStart[v] >
                  WindowEnd[k]) then
17            impossibleMove ← true
18          end if
19        end if
20      end for
21      if not impossibleMove then
22        legalMovesPerStep[step].append(v)
23      end if
24    end if
25  end for
26 end if
27 if there are no moves in legalMovesPerStep[step]
      then
28  for v from 1 to V do
29    if not visited[v] then
30      legalMovesPerStep[step].append(v)
31    end if
32  end for
33 end if
```

Algoritmo 4. Calcula los movimientos permitidos que pueden hacer desde *u* en el paso *step*

```
1 (r Rollout) pickMove(u, step, policy, moves)
2  z ← 0
3  for m in moves[step] do
4    v := moves[step][m]
5    moveProb[m] ← e^policy[u][v]
6    z ← z + moveProb[m]
7  end for
8  idx ← 0
9  random ← randGenerator() * z
10 probAcc ← moveProb[idx]
11 while probAcc < random do
12   idx ← idx + 1
13   probAcc ← probAcc + moveProb[idx]
14 end while
15 return moves[step][idx]
```

Algoritmo 5. Selecciona un movimiento de forma aleatoria de acuerdo con la distribucion de la politica

```
1 (r *Rollout) move(u, v)
2 r.Tour[r.Length] ← v
3 r.Length ← r.Length + 1
4 visited[v] ← true
5 r.cost ← r.cost + Distances[u][v]
6 r.Makespan ← Max(r.Makespan+Distances[u][v],
      WindowStart[v])
7 if r.Makespan > WindowEnd[v] then
8   r.Violations++
9 end if
```

Algoritmo 6. Realiza un movimiento de u → v

III-C. SNRPA

Stabilized Nested Rollout Policy Adaptation (SNRPA) es una versión modificada de NRPA propuesta por Tristan Cazenave [1]. La modificación consiste en no adaptar la política en el nivel más bajo de la recursión ya que las políticas los mejores *rollouts* encontrados en este nivel dependientes completamente de la probabilidad y seguramente no contienen conocimiento refinado que guie la búsqueda a mejores casos, además el nivel superior se va a ver afectado ya que si por casualidad, el nivel 0 genera constantemente *rollouts* con puntajes malos, el nivel 1 va a adaptar la política con su mejor *rollout* lo cual hace la búsqueda más determinística y restringe la exploración de mejores *rollouts*, este caso se propaga a los niveles superiores y afecta negativamente todo el árbol de búsqueda.

En lugar de aptar el nivel 1 se van a generar *F rollouts* con la política del nivel 2 sin modificarla, el mejor de estos generado de forma aleatoria se usa para adaptar la política del nivel 2

```
1 SNRPA(level, policy)
2  level.bestRollout.score ← −∞
3  if level.id == 1 then
4    for F iterations do
5      bestRollout ← playout(policy)
6      if bestRollout.score ≥ level.bestRollout.
           score then
7        level.bestRollout ← bestRollout
8        level.movesPerStep ← nextLevel.
             movesPerStep
9      end if
```

```
10  else
11   level.policy ← policy
12   for N iterations do
13     nextLevel ← DataPerLevel[level.id − 1]
14     bestRollout ← SNRPA(nextLevel, level.policy)
15     if bestRollout.score ≥ level.bestRollout.score
           then
16       level.bestRollout ← bestRollout
17       level.movesPerStep ← nextLevel.movesPerStep
18     end if
19     level.AdaptPolicy()
20   end for
21   return level.bestRollout
22  end if
```

Algoritmo 7. El algoritmo SNRPA

El *algoritmo 7* es muy similar a NRPA, de hecho las líneas 10-22 son casi todo NRPA. SNRPA facilita la implementación de soluciones concurrentes usando concurrencia en el cálculo de las hojas del árbol de recursión durante el proceso de la generación del *rollout*, porque al no modificarse la política esta se puede compartir y hace que el cálculo de cada hola del nivel sea independiente, por lo que se puede realizar sin mucho esfuerzo de forma concurrente.

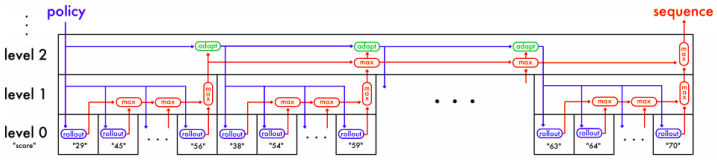

Figura 2. Representación de SNRPA tomado de [1]

IV. Optimizaciones

IV-A. Concurrencia

Con el fin de sacar todo el partido de los procesadores multinúcleo y los clústeres de cómputo, se busca paralelizar el computo de SNRPA para esto se propone un modelos de actores. Un actor se puede definir como una estructura que se confirma de un conjunto de datos y la capacidad de ejecutar acciones de forma concurrente, dependiendo de la abstracción del lenguaje para la concurrencia se puede ver como datos y un hilo o rutina.

El modelo está formado por dos tipos de actores. El primer grupo se encarga de calcular los *rollout*, estos poseen la información necesaria del tablero y la información auxiliar necesaria para calcular un *rollout* como una lista de visitados, movimientos legales, y probabilidades de hacer un movimiento, con esto la información permanece local al actor al momento de calcular el *rollout* evitando el acceso compartidos a recursos cambiantes entre ejecuciones concurrentes, lo cual suele llevar a condiciones de carrera. Además facilita controlar el timeout del programa al detener los actores y permiten manejar el nivel de concurrencia del programa con solo cambiar la cantidad de actores que se generan.

El segundo grupo de actores se va a encarga de calcular un árbol entero de SNRPA, estos actores van a tener la información necesaria para trabajar de forma independiente y van a crear actores del otro grupo, los van a usar para calcular las hojas de su árbol. Estos son necesarios porque es normal correr más de una instancia de un árbol SNRPA para reducir las probabilidades de tener mala suerte en la ejecución de una busqueda y terminar con un resultado mediocre. Al número de actores que procesan un árbol la llamaremos *nActores* y al número de actores que procesan las hojas lo llamaremos *pActores* por defecto estas dos son iguales a el número de *runs* (cantidad de árboles a correr) y *F* el factor estabilizador de SNRPA.

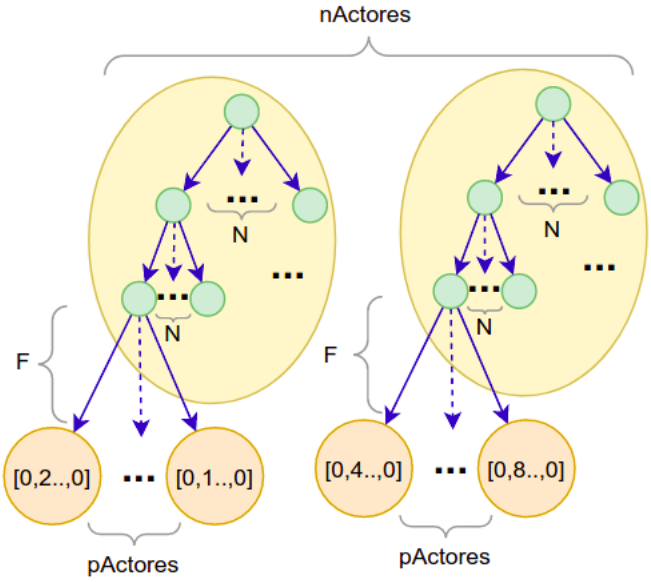

Figura 3. Modelo de actores

IV-B. Optimizaciones de la implementación

Aunque por lo general se pone mucho esfuerzo en el diseño conceptual de soluciones haciendo uso de técnicas de análisis de algoritmos, la implementación de dichas soluciones no es revisada con tanto detalle y esto puede dejar mucho desempeño desperdiciado, por eso en esta sección se cubrirán algunas de las optimizaciones que permitieron acelerar un 300 % la primera versión de la implementación.

IV-B1. Profiler: Con la ayuda de pprof se identificaron los cuellos de botella que ralentizaban la ejecución de una primera ejecución se obtuvo:

flat	flat %	cum	cum %	func
28.69s	71.00 %	28.71s	71.05 %	alda/nrpa.(*Rollout).calculateLegalMoves
4.38s	10.84 %	4.38s	10.84 %	math.Exp
1.79s	4.43 %	6.26s	15.49 %	alda/nrpa.(*Rollout).pickMove
0.77s	1.91 %	5.15s	12.74 %	alda/utils.Exp
0.33s	0.82 %	0.33s	0.82 %	runtime.futex
0.32s	0.79 %	1.33s	3.29 %	alda/nrpa.(*Level).AdaptPolicy

Cuadro I
ANALISIS USO DE CPU TOP10 FUNCIONES

De estos datos se puede analizar que la mayor parte del tiempo se usa en el cálculo de los movimientos validos de un estado, por lo que se puso especial cuidado a todas las asignaciones de memoria y comparaciones para simplificarlas,

prealocar la matriz de movimientos por actor mejoró los tiempos en un 10 %, simplificar una expresión booleana y evitar usar math.Max (de Golang) mejoraron un 60 % más el rendimiento, ya que math.Max hace validaciones para valores especiales con ∞ y *NaN* la hacen considerablemente más lenta que solo comparar los dos valores.

La segunda función con mayor consumo es el cálculo del exponencial, por lo que lo remplazaremos con una aproximación que no sea tan precisa pero si mucho más rápida, dado que e^x se define por el límite como:

$$e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n$$

Podemos aproximar e^x si evaluamos el límite con un valor n fijo 1024.

```
1  Exp(x)
2    x ← 1.0 + x/1024
3    x ← x * x
4    x ← x * x
5    x ← x * x
6    x ← x * x
7    x ← x * x
8    x ← x * x
9    x ← x * x
10   x ← x * x
11   x ← x * x
12   x ← x * x
13   return x
```

Algoritmo 8. Aproximación de la función exponencial con n

La multiplicación de $x * x$ acumulada 8 veces equivale a la exponenciación y por propiedades de los exponentes $a^{x^x} = a^{x*x}$, entonces $2^8 = 1024$ podemos calcular en 8 multiplicaciones el exponente.

Por último, llama la atención la cantidad de llamados en bloqueos para mantener la atomicidad cuando no hay información compartida, pero mirando en detalle eso sucede porque rand.Float64 es un función segura para usar de forma concurrente, por lo que bloquea en cada llamado, para solucionar esto usamos un generado independiente por actor que calcula las hojas del árbol.

V. Resultados

Los siguentes resultados fueron obtenidos en ordenador con las siguentes especificaciones CPU Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, memoria DDR4 2400MHz 16GiB, SO Ubuntu 20.04.2 LTS (5.4.0-66-generic)

VI. Conclusiones y trabajo futuro

SNRPA es un algortmo que facilmente puede computarse de forma concurrente por lo que permite sacar probecho de los CPUs multicore y sistemas de computo distribuido en clusters, el algortmo es capaz de encontrar una solución valida para todos los casos, en tiempos inferiores a 1 min, aunque en alguna situaciónes no se ecnuentra tan cerca del optimo como se desearía, en alguno de esos casos ya es dificil encontrar un camino que no incumpla las ventana de tiempo.

Como trabajo futuro explorar GNRPA e implementar una versión estabilizada y paralera de este, ya que promete a travez del uso de un gradiente mejorar considerablemente los resultados, especialmente en casos dicifiles como el 204.1

problema	tiempo	$\phi(T)$	puntaje	makespan
rc_201.1.txt	1.00s	0	444.54	592.06
rc_201.2.txt	1.00s	0	712.91	869.90
rc_201.3.txt	1.00s	0	795.44	854.12
rc_201.4.txt	1.00s	0	793.64	889.18
rc_202.1.txt	1.00s	0	790.04	850.48
rc_202.2.txt	1.00s	0	305.76	338.52
rc_202.3.txt	1.00s	0	856.64	894.10
rc_202.4.txt	1.00s	0	793.03	854.12
rc_203.1.txt	1.00s	0	453.48	488.42
rc_203.2.txt	1.00s	0	836.63	853.71
rc_203.3.txt	1.00s	0	858.60	941.36
rc_203.4.txt	1.00s	0	320.60	338.52
rc_204.1.txt	1.00s	13	13001390.25	1390.25
rc_204.2.txt	1.00s	0	817.89	845.79
rc_204.3.txt	1.00s	0	472.58	700.53
rc_205.1.txt	1.00s	0	343.21	455.94
rc_205.2.txt	1.00s	0	755.93	820.19
rc_205.3.txt	1.00s	0	880.02	951.67
rc_205.4.txt	1.00s	0	776.41	853.85
rc_206.1.txt	1.00s	0	117.85	117.85
rc_206.2.txt	1.00s	1	1000900.90	901.11
rc_206.3.txt	1.00s	0	583.36	661.07
rc_206.4.txt	1.00s	0	900.41	929.85
rc_207.1.txt	1.00s	0	826.08	901.10
rc_207.2.txt	1.00s	2	2000878.68	904.45
rc_207.3.txt	1.00s	0	774.91	798.66
rc_207.4.txt	1.00s	0	119.64	214.50
rc_208.1.txt	1.00s	13	13001159.85	1163.23
rc_208.2.txt	1.00s	0	646.36	646.36
rc_208.3.txt	1.00s	1	1000858.60	871.00

Cuadro II
Resultados con timeout de 1s

problema	tiempo	$\phi(T)$	puntaje	makespan
rc_201.1.txt	10.00s	0	444.54	592.06
rc_201.2.txt	10.00s	0	712.91	869.90
rc_201.3.txt	10.00s	0	793.75	854.12
rc_201.4.txt	10.00s	0	793.64	889.18
rc_202.1.txt	10.00s	0	775.72	850.48
rc_202.2.txt	9.46s	0	305.76	338.52
rc_202.3.txt	10.00s	0	858.30	894.10
rc_202.4.txt	10.00s	0	793.03	854.12
rc_203.1.txt	10.00s	0	453.48	488.42
rc_203.2.txt	10.00s	0	785.34	853.71
rc_203.3.txt	10.00s	0	828.17	955.12
rc_203.4.txt	10.00s	0	320.60	338.52
rc_204.1.txt	10.00s	7	7001149.79	1149.79
rc_204.2.txt	10.00s	0	706.89	835.74
rc_204.3.txt	10.00s	0	460.47	630.94
rc_205.1.txt	6.81s	0	343.21	455.94
rc_205.2.txt	10.00s	0	755.93	820.19
rc_205.3.txt	10.00s	0	847.95	950.05
rc_205.4.txt	10.00s	0	769.55	853.85
rc_206.1.txt	3.92s	0	117.85	117.85
rc_206.2.txt	10.00s	0	868.29	917.26
rc_206.3.txt	10.00s	0	578.09	661.07
rc_206.4.txt	10.00s	0	855.13	924.91
rc_207.1.txt	10.00s	0	746.30	857.95
rc_207.2.txt	10.00s	0	720.01	732.03
rc_207.3.txt	10.00s	0	690.88	790.54
rc_207.4.txt	4.21s	0	119.64	214.50
rc_208.1.txt	10.00s	1	1000951.17	951.17
rc_208.2.txt	10.00s	0	562.49	638.16
rc_208.3.txt	10.00s	0	770.35	770.35

Cuadro III
Resultados con timeout de 10s

problema	tiempo	$\phi(T)$	puntaje	makespan
rc_201.1.txt	11.18s	0	444.54	592.06
rc_201.2.txt	15.85s	0	712.91	869.90
rc_201.3.txt	27.77s	0	795.44	854.12
rc_201.4.txt	16.17s	0	793.64	889.18
rc_202.1.txt	40.74s	0	775.89	850.48
rc_202.2.txt	9.36s	0	305.76	338.52
rc_202.3.txt	23.08s	0	856.64	894.10
rc_202.4.txt	28.26s	0	793.03	854.12
rc_203.1.txt	15.75s	0	453.48	488.42
rc_203.2.txt	50.16s	0	785.34	853.71
rc_203.3.txt	60.00s	0	819.42	955.12
rc_203.4.txt	11.17s	0	320.60	338.52
rc_204.1.txt	60.00s	0	943.20	950.36
rc_204.2.txt	60.00s	0	675.18	727.39
rc_204.3.txt	36.37s	0	455.03	455.03
rc_205.1.txt	6.74s	0	343.21	455.94
rc_205.2.txt	22.41s	0	755.93	820.19
rc_205.3.txt	50.36s	0	843.04	950.05
rc_205.4.txt	24.15s	0	762.32	853.85
rc_206.1.txt	3.99s	0	117.85	117.85
rc_206.2.txt	54.06s	0	844.62	917.26
rc_206.3.txt	25.63s	0	574.42	661.07
rc_206.4.txt	54.39s	0	838.89	930.09
rc_207.1.txt	50.17s	0	749.28	857.62
rc_207.2.txt	45.43s	0	710.74	728.12
rc_207.3.txt	51.91s	0	695.87	811.75
rc_207.4.txt	4.31s	0	119.64	214.50
rc_208.1.txt	60.00s	0	857.74	906.89
rc_208.2.txt	53.19s	0	570.13	644.95
rc_208.3.txt	60.00s	0	711.17	786.85

Cuadro IV
RESULTADOS CON TIMEOUT DE 60S

REFERENCIAS

[1] Cazenave, T., 2021. Stabilized Nested Rollout Policy Adaptation (CG 2021). [online] pp.1-15. Available at: ¡https://arxiv.org/pdf/2101.03563.pdf¿[Accessed 6 March 2021].

[2] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1), 1–43 (Mar 2012). https://doi.org/10.1109/TCIAIG.2012.2186810

[3] Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; Driessche, George van den; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda; Lanctot, Marc; Dieleman, Sander; Grewe, Dominik; Nham, John; Kalchbrenner, Nal; Sutskever, Ilya; Lillicrap, Timothy; Leach, Madeleine; Kavukcuoglu, Koray; Graepel, Thore; Hassabis, Demis (28 January 2016). "Mastering the game of Go with deep neural networks and tree search". Nature. 529 (7587): 484–489.

[4] Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: IJCAI. pp. 649–654 (2011)