Simmulated Annealing en Travelling Tournament Problem

Yoel Berant Elorza, Universidad Técnica Federico Santa María

***Abstract*—En este informe se propone un algoritmo Simmulated Annealing para resolver Travelling Tournament Problem, un problema que consiste en organizar un torneo entre varios equipos que satisfaga muchas restricciones y que, al mismo tiempo, minimice la distancia de viaje que todos los equipos deban recorrer para jugar los partidos. Gran parte de este informe consiste en explicar este problema y explorar algunas soluciones ya propuestas.**

Previo a este informe se realizó la evaluación de otro método implementado, basado esta vez en la heurística de Greedy Search. En este informe además se realizará una comparación entre los resultados obtenidos usando el método Simmulated Annealing propuesto y el algoritmo Greedy.

1. Introducción

El "travelling tournament problem" (o TTP) [7] es un problema de optimización complejo, el cual consiste en programar los partidos que se jugarán durante el torneo de algún deporte. En este torneo, cada equipo debe competir contra los demás equipos dos veces: una jugando como local y la otra como visita.

El objetivo consiste en organizar los partidos del torneo tal que se minimice la distancia que los equipos tengan que recorrer, además de que se cumplan varias restricciones.

En este documento se recopila información sobre el TTP, incluyendo una descripción detallada de este problema, el contexto en el que nace, algunas soluciones propuestas, se propone y se implementa un método Simmulated Annealing para entregar una solución al problema. Finalmente, se comparan los resultados con los entregados por otra implementación del tipo Greedy Search realizada previamente.

2. Estado del arte

A. Contexto

El entretenimiento a través del deporte es una industria muy lucrativa en todo el mundo. Cada año, eventos deportivos televisados mueven suficiente dinero como para generar un impacto significativo en las economías de varios países. Tan solo por poner un ejemplo, el 2019, solo la final de uno de los torneos de Fútbol más importantes del mundo: la "Champions League", realizada en Madrid, España, generó un impacto económico de 123 millones de Euros, de los cuales al menos 66 se quedaron en la capital del país ibérico, específicamente en los sectores de ocio, hotelero, gastronómico y de compra de souvenires [5].

Por esto mismo, existen organizaciones e incluso gobiernos capaces de invertir millonadas en este tipo de eventos. Es bien sabido, dando otro ejemplo, que las olimpiadas del 2016, realizadas en Río de Janeiro, costaron cerca de 13 mil millones de dolares, de los cuales gran parte vinieron del gobierno de Brasil [15].

Dejando de lado la construcción y mantención de estadios, la publicidad de los eventos y la remuneración que reciben los equipos participantes y sus agencias, parte de todo este dinero también se debe invertir en los viajes que los equipos deben realizar.

Los torneos **"round robin"** (RR), son torneos en los que cada equipo participante debe jugar contra el resto de equipos una sola vez, participando simultáneamente todos los equipos una vez por ronda, existiendo $N-1$ rondas (donde N es el número de equipos). A su vez, en los torneos **"double round robin"** (DRR), cada equipo que participa debe jugar no una, sino dos veces contra cada uno de los otros equipos; esta vez, todos los equipos participan simultáneamente en cada una de las $2*(N-1)$ rondas. Un caso especial de torneos DRR ocurre cuando en cada par de equipos, en uno de los dos partidos un equipo debe jugar como local y el otro de visita, mientras que en el otro partido, el que jugaba como local juega como visita y vise versa.

No es difícil darse cuenta que si un torneo corresponde al caso especial de DDR recién explicado (el cual en este informe se hará referencia cada vez que se mencionen a los torneos DRR de ahora en adelante), se vuelve extremadamente importante una buena organización del orden en el que se jugarán los partidos, con el fin de minimizar la inversión en viajes (considerando que los precios de los viajes varían según la distancia que se tiene que recorrer). Es en este contexto en el que nace **Travelling Tournament Problem**, propuesto por Kelly Easton, George Nemhauser y Michael A. Trick. [7] Se trata de un problema de optimización que plantea la necesidad de planear los partidos de un torneo DRR de esta categoría con el fin de optimizar el costo total de los viajes, además de cumplir con ciertas restricciones.

B. Descripción del problema

TTP consiste en lo siguiente:

Se tiene un número par N de equipos, los cuales jugarán en un torneo DRR, es decir, cada par de equipos debe jugar un partido dos veces: en uno de esos partidos un equipo jugará de visita y el otro de local, mientras que en el otro partido se invertirán los roles de los equipos. Por ejemplo, si en un partido, el equipo A juega como visita contra el equipo B, que

juega local, debe jugarse otro partido entre A y B, en el que A juegue como local y B como visita.

Cada equipo entrena en su "casa" propia. Una instancia del problema corresponde a una matriz numérica NxN que representa las distancias entre las casas de los equipos. Si un equipo juega como local en un partido, debe jugarlo desde su casa (en ese caso su contrincante jugaría como visita) y si juega como visita, debe viajar a la casa su contrincante (el cual estaría jugando como local).

El torneo consiste en $2 * (N - 1)$ rondas. En cada ronda, juegan simultáneamente todos los N equipos en $N/2$ partidos. El objetivo es escoger los partidos que se jugarán en cada ronda, tal que se minimice la distancia total que todos los equipos deban recorrer entre rondas, considerando también que cada equipo se encuentra inicialmente en su casa y que al final del torneo, cada equipo debe volver esta (estos últimos viajes se cuentan en el costo)

Además, deben cumplirse las siguientes restricciones añadidas:

- **atmost:** Ningún equipo puede jugar como local en más de $U = 3$ rondas consecutivas o como visita en $U = 3$ rondas consecutivas. (Nota: Cuando se propuso TTP por primera vez [7], el valor de U podía variar según la instancia y además podía existir un número mínimo L de juegos de visita o local consecutivos por equipo, pero típicamente se fija $L = 1$ y $U = 3$)
- **norepeat:** Ningún par de equipos puede competir entre sí en dos rondas seguidas.

C. Soluciones Planteadas

A continuación, se describen brevemente algunos métodos planteados anteriormente para resolver TTP, junto con sus resultados.

1) Solución Original: Kelly Easton, George Nemhausery Michael A. Trick, las mismas personas que definieron TTP [7] propusieron un método para resolverlo basado en constraint programming, integer programming y branch and bound [8].

La motivación detrás de esta decisión en cuanto a paradigmas [8] es que TTP es un problema tanto de optimización (pues se busca minimizar la distancia total de viajes de cada equipo) como de satisfacción de restricciones (como "atmost", ver sección 2.B). De este modo, constraint programming se encargaría de que la solución final respeta las restricciones e integer programming, de mimimizar la función objetivo.

El primer paso es, para cada equipo, definir un patrón de juegos visita-local. Por ejemplo, para $N = 6$ ($2 * (N - 1) = 10$ rondas), un patrón sería $\{L, L, V, V, V, L, L, L, V, V\}$. Nótese que según este patrón, el equipo no juega más de 3 partidos seguidos del mismo "tipo". Para realizar este paso se hace uso de constraint programming, es decir, ir asignando los patrones a los equipos con cuidado de que no se infrinjan restricciones.

El siguiente paso es definir los tours para cada equipo (orden de contrincantes en cada ronda) P, que resuelva el siguiente problema:

Minimizar: $\sum_{i \in P} c_i x_i$, sujeto a:

- $\sum_{i \in P_t} x_i = 1 \ ,\forall t \in T$
- $\sum_{\{i \in P : i \notin P_t \text{ y } t \text{ es un oponente en la ronda } r \text{ dentro de } i\}} x_i + \sum_{\{i \in P_t : t \text{ es visita en la ronda } r \text{ en } i\}} x_i = 1 \ ,\forall r \in R, \ t \in T, \ i \in P.$

Donde x_i es una variable binaria que indica si el tour i se incluye, c_i es el costo (o distancia total recorrida) del tour i, R es el conjunto de rondas y T, el conjunto de equipos.

Para resolver este problema se utiliza integer programming, usando una relajación en la restricción de variables enteras (se pueden obtener variables no enteras) y luego branch and bound, es decir, en cada solución con valores no enteros, fijar un valor no entero redondeándolo hacia arriba y hacia abajo y reoptimizar en cada caso. Para acelerar el proceso, se usa programación paralela (varios threads) mediante un paradigma "maestro-esclavo".

Este método encontró soluciones de distancia total muy cercana a la óptima en menos de un minuto para instancias de $N = 4$, en alrededor de 15 minutos para $N = 6$ y varios días para soluciones de $N = 8$. En otras palabras, el algoritmo es muy eficaz pero poco eficiente.

2) Simmulated Annealing: En el año 2003, A. Anagnostopoulos, L. Michel, P. Van Hentenryck, e Y. Vergados [1] propusieron resolver TTP usando Simmulated Annealing (SA).

SA [13] es una heurística de búsqueda incompleta de soluciones, la cual consiste en generar una solución inicial e ir transformándola, realizando permutaciones.

Una transformación o movimiento es aceptada si el resultado es mejor que la solución actual (según una función de evaluación, la cual se detallará más adelante). En caso contrario, puede ser aceptada según una variable aleatoria que depende de que tan "no mejor" sea la transformación y de una "temperatura" (la cual puede ir variando entre permutaciones).

En este caso, los autores de esta implementación [1] argumentan que las siguientes características de SA permitirían encontrar soluciones óptimas para TTP más facilmente:

- SA da la posibilidad de explorar tanto soluciones factibles como infactibles, al relajar algunas restricciones (más adelante en esta sección se explicará esto)
- El vecindario de movimientos para cada iteración es muy grande ($O(N^3)$), lo que significa una mayor exploración en el espacio de soluciones. Además, algunos movimientos son muy complejos y capaces de cambiar la solución actual en maneras significativas.
- La implementación puede incluir estrategias para balancear el tiempo gastado en explorar regiones factibles e infactibles.
- La implementación puede incorporar "recalentamientos", es decir, reiniciar la variable de temperatura con el fin de evitar un estancamiento en óptimos locales.

En este caso, cada solución se representa como una tabla, en donde cada fila corresponde a un equipo y cada columna una ronda. El valor en cada casilla indica contra que debe jugar el equipo de la fila en la ronda de la casilla. Un valor

positivo indica que se tendrá que el equipo (de la fila) jugará como local, mientras que un valor negativo indica que jugará como visita.

	r_1	r_2	r_3	r_4	r_5	r_6
T_1	$-T_3$	T_2	T_3	$-T_4$	$-T_2$	T_4
T_2	T_4	$-T_1$	$-T_4$	$-T_2$	T_1	T_2
T_3	T_1	$-T_4$	$-T_1$	T_3	T_4	$-T_3$
T_4	$-T_2$	T_3	T_2	T_1	$-T_3$	$-T_1$

En el ejemplo de arriba ($N = 4$), el equipo T_1 debe jugar de visita contra T_3, de local contra T_2, de local contra T_3, de visita contra T_4, de visita contra T_2 y de local contra T_4, en ese orden.

En cuanto a las permutaciones, se implementan 5 tipos de movimientos:

1) $SwapHomes(T_i, T_j)$: Encuentra los 2 juegos en los que T_i compite contra T_j, e intercambia los roles visita-local. En la tabla, sería equivalente a encontrar las casillas en las que aparece $\pm T_i$ en la fila de T_j y vice versa, y cambiarles los signos.
2) $SwapRounds(r_k, r_l)$: Intercambia los juegos asignados en las rondas r_k y r_l. En la tabla, sería equivalente a intercambiar las columnas de las rondas r_k y r_l.
3) $SwapTeams(T_i, T_j)$: Intercambia los partidos programados para los equipos T_i y T_j, exceptuando a los partidos en que juegan T_i contra T_j. En la tabla, sería equivalente a intercambiar las filas de T_i y T_j, excepto las casillas en que aparecen $\pm T_i$ y $\pm T_j$; además, se actualizan casillas afectadas (en donde otros equipos jueguen contra T_i y T_j).
4) $PartialSwapRounds(T_i, r_k, r_l)$: Intercambia los partidos jugados por T_i en las rondas r_k y r_l. Luego, realiza los intercambios de partidos en r_k y r_l necesarios (de manera determinista [2]) para que los equipos jueguen una y solo una vez en cada una de esas rondas.
5) $PartialSwapTeams(r_k, T_i, T_j)$: Similar a $PartialSwapRounds$, pero esta vez intercambia los partidos jugados por T_i y T_j en la ronda r_k, y luego hace los intercambios de partidos necesarios entre T_i y T_j (de manera determinista [2]) para que cada equipo juegue contra el resto de equipos una vez como local, y otra como visita.

Es preciso señalar que las restricciones atmost y norepeat se relajan, formando parte de la función de evaluación. Es decir, a parte de minimizar la distancia recorrida se intentan minimizar también las infracciones a estas restricciones, incluyendo el numero de infracciones en la función de evaluación que se busca minimizar. La función de evaluación sería entonces:

$$C(S) = \sqrt{dist(S)^2 + (w * (1 + \frac{\sqrt{v} \ln v}{2}))^2}$$

Donde $dist(S)$ es la distancia total recorrida por todos los equipos en el torneo S y v es el número de infracciones a las restricciones atmost y norepeat.

Esto implica no rechazar soluciones en que ocurran estas infracciones (de todos modos, las soluciones finales tienen que respetar estar restricciones. Esto aplica para los otros métodos que se mencionarán en este informe de ahora en adelante que también relajen estas restricciones).

Los resultados de la implementación fueron bastante favorables para la época. Se experimentó usando instancias del benchmark "National League" [7], cada una con un N distinto, y en cada una se experimentó 50 veces (variando hiperparámetros y considerando aleatoriedad). Los resultados superaron a los mejores resultados obtenidos en esa época (la implementación se hizo en 2003, pero los "mejores resultados" son del 2002).

n	Best (Nov. 2002)	min(D)	max(D)	mean(D)	std(D)
8	39721	39721	39721	39721	0
10	61608	**59583**	**59806**	**59605.96**	53.36
12	118955	**112800**	**114946**	**113853.00**	467.91
14	205894	**190368**	**195456**	**192931.86**	1188.08
16	281660	**267194**	**280925**	**275015.88**	2488.02

Fig. 1. Resultados de implementación de Simmulated Annealing, realizada por A. Anagnostopoulos, L. Michel, P. Van Hentenryck, e Y. Vergados en el año 2003[1]. En la tabla, se ve la distancia total recorrida por equipo mínima, máxima, promedio y su desviación estandar alcanzadas en cada instancia del benchmark National League

En cuanto a los tiempos de ejecución, el algoritmo demora de 10 minutos a varios días en completarse, dependiendo del tamaño de N, según la siguiente tabla:

n	min(T)	mean(T)	std(T)
8	596.6	1639.33	332.38
10	8084.2	40268.62	45890.30
12	28526.0	68505.26	63455.32
14	418358.2	233578.35	179176.59
16	344633.4	192086.55	149711.85

Fig. 2. tiempos de ejecución en segundos de las implementaciones de simmulated annealing [1]

3) Simmulated Annealing con coloreo de grafo: En el año 2012, Sevnaz Nourollahi, Kourosh Eshghi y Hooshmand Shokri Razaghi [12] implementan una variación del método de SA propuesto en 2003 [1].

Esta vez, se basan en que la representación de soluciones de TTP, al igual que cualquier representación de torneos DRR puede ser equivalente al coloreo de un grafo.

En este grafo, cada vértice representaría un posible juego o partido, representado por el par $\{i, j\}$, en el que i juega como local e j como visita. Los ejes representan la imposibilidad de que pares de partidos (es decir, los dos vértices unidos por cada eje) se jueguen en la misma ronda. Esto se da en el caso de que si los dos vértices son $\{i, j\}$ y $\{k, l\}$, se de el caso de que i o j sea igual a k o a l[10].

Por ejemplo, los vértices $(1, 3)$ y $(8, 1)$ están unidos, porque los dos comparten al equipo 1, y el equipo 1 no puede jugar contra el equipo 3 y el equipo 8 en la misma ronda.

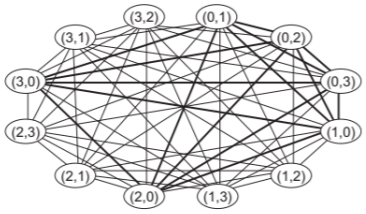

Fig. 3. representación de juegos de un DRR a través de un grafo, con N=4

De esta forma, si cada color es una ronda, aplicando un coloreo de grafos (en que dos vértices adyacentes no puedan ser pintados con el mismo color) quedaría un DRR, siempre y cuando se imponga la restricción de que deben haber exactamente $2 * (n - 1)$ colores en el coloreo.

Usando esta representación, se propone añadir a la implementación de SA anteriormente explicada un sexto tipo de movimiento: $KempeChainMove(T_i, r_k, r_l)$, el cual consiste en encontrar una *cadena kempe*[10], es decir, una cadena (secuencia de vértices conectados) contenida en la solución actual en la cual los nodos estén pintados de solo dos colores alternantes, e intercambiar esos los colores.

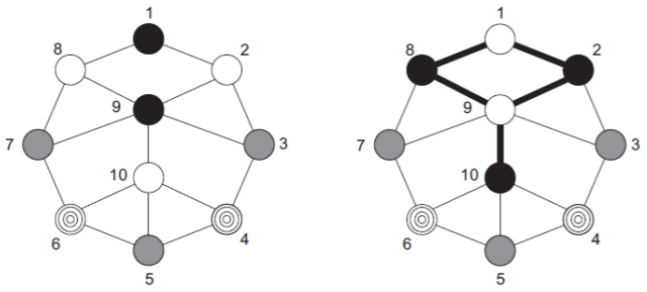

Fig. 4. ejemplo de intercambio de colores en cadena kempe[12].

En este caso, se busca una cadena que parta desde el vértice que represente el partido en el que juega el equipo T_i en la ronda (color) r_k e incluya vértices del color de $r_l \neq r_k$ (la cadena tendría un total de 4 vértices). Al intercambiar los colores de esa cadena, se realiza un cambio sin violar las restricciones duras de DRR.

En palabras simples, lo que hace este movimiento es intercambiar el partido de la ronda r_k en el que juega T_i, y algún partido de la ronda r_l sin que se violen las restricciones: "todos los equipos juegan una y solo una vez por ronda" y "todos los equipos juegan contra todos, una vez como local y otra como visita". (a diferencia de los movimientos PartialSwapRounds y PartialSwapTeams, en los cuales era necesario realizar cambios adicionales luego de intercambiar dos partidos para satisfacer estas restricciones).

Otra modificación importante que incorpora esta implementación es que tanto este movimiento, como "PartialSwapTeams" tienen mayor probabilidad de ser escogidos durante cada iteración. La motivación detrás de esta mejora, es que ambos movimientos son menos triviales que los demás y, al ser escogidos con más frecuencia, permiten una exploración más rápida del espacio de búsqueda [12].

Para probar esta implementación se usaron instancias del benchmark "Major League Baseball" [4]. En este caso, se usaron instancias de $N = 4$, $N = 6$, $N = 8$ y $N = 10$ (una instancia por valor de N). Exceptuando el caso de $N = 10$, se tenía conocimiento previo de las soluciones óptimas, las cuales fueron alcanzadas en esta implementación. En la siguiente tabla, se comparan los resultados de esta implementación (*Efficient SA*) con otras los resultados de otras implementaciones (SA, LR-CP, SA-Hill, PSO-SA) que también se basan en SA, tanto en mejor solución encontrada (distancia total recorrida) encontrada (BFS) como en tiempo de ejecución, en segundos (TST):

Method		NL4	NL6	NL8	NL10
SA	BFS	-	-	39721	59583
	TRT	-	-	1639	40269
LR-CP	BFS	8276	23916	42517	68691
	TRT	1.5	86400	14400	86400
SA-Hill	BFS	8276	23916	39721	59821
	TRT	1.7	821	4107	40289
PSO-SA	BFS	8276	23916	39721	65002
	TRT	0.2	30	1800	7200
Efficient SA	BFS	8276	23916	39721	61956
	TRT	0.0	0.0	667	3000

Como se puede ver, la implementación [12] alcanza mejores soluciones en un tiempo menor a las demás.

4) Algoritmos genéticos con Hadoop: Laxmi Thakare, Dr. Jayant Umale y Bhushan Thakare [14] proponen en el año 2012 una implementación basada en algoritmos genéticos (GA).

Los GA [6] son métodos de búsqueda basados en los procesos evolutivos y de selección natural presentes en los organismos de la vida real. Los GA seleccionan la mejor de muchas soluciones generadas a traves de los siguientes pasos:

1) Inicializar una generación de soluciones $P(t)$.
2) Elegir de $P(t)$ algunas soluciones o "cromosomas". Los mejores cromosomas (según alguna función objetivo) suelen tener mas probabilidad de ser escogidas (elitismo).
3) Combinar y/o transformar cromosomas, verificando si las funciones objetivo mejoren.
4) Generar siguiente $P(t+1)$, insertando tanto cromosomas aleatorios como los mejores cromosomas obtenidos de los pasos 2 y 3.
5) $t = t + 1$
6) Volver varias veces al paso 2, hasta que se hayan realizado las iteraciones suficientes.
7) Devolver el mejor cromosoma (solución) encontrado durante los pasos anteriores.

No es difícil adivinar que los algoritmos genéticos involucran procesar una cantidad muy alta de datos. Por lo tanto, esta implementación se usó **Hadoop**, un framework de PHP especializado en procesos **mapreduce**.

Un proceso mapreduce consiste en dos etapas: el "mapeo", donde se aplica una función sobre cada uno de los datos y la "reducción", donde se combinan los resultados del mapeo en una sola función.

Para realizar un trabajo más eficiente durante el mapeo, Hadoop divide los datos en varios bloques y aplica programación paralela sobre cada bloque.

Hadoop intuitivamente puede resultar muy útil para GA, puesto que se pueden aplicar procesos mapreduce para comparar y seleccionar varias soluciones en una generación, y para mutar y combinar soluciones seleccionadas.

La motivación detrás de esta implementación, según argumentan sus autores [14], es debido al gran tamaño de la población que exploran GA, el gran número de iteraciones y el paralelismo intrínseco de esta heurística, factores que aumentarían la probabilidad de alcanzar una solución globalmente óptima. Con respecto al tiempo de ejecución que demanda GA (el cual suele ser la razón de porque se descarta el uste método), este sería reducido significativamente gracias a la eficiente tecnología de programación paralela y mapreduce que Hadoop brinda.

Para la implementación se consideraron las instancias del benchmmark "National League" [7] de $N = 4$, $N = 6$ y $N = 8$. En la siguiente tabla, se comparan los resultados obtenidos (TCT) y el tiempo de ejecución (TT) entre una implementación de GA usando Hadoop y mapreduce y otra implementación de hecha en java (ambas implementaciones fueron desarrolladas por Thakare, Umale y Thakare:

Instance	CL	TT		TCT			Observation
		Java	MR	Feasible Range[15]	Actual Result(java)	Actual Result(MR)	
NL-4	12	00:17	0:15	8276	8276	8276	Optimal
NL-6	30	04:54	03:37	22969-23916	24349	23861	Optimal
NL-8	56	04:39:47	01:46:49	39721-41505	40972	41285	Optimal

Analizando la tabla, el uso de mapreduce resulta bastante útil para algoritmos que implican procesar muchos datos como éste, aumentando la eficiencia sin sacrificar la eficacia (puesto que para las tres instancias se llegó a una solución óptima). Sin embargo, si consideramos que la instancia de National League con $N = 8$ también se probó en la implementación de SA para resolver TTP (ver sección 2.C.2), al comparar los resultados podemos ver que SA es más rápido y más eficaz, pues SA fue capaz de encontrar una solución óptima (39721) en menos de una hora, mientras que GA mapreduce demoró más de una hora en encontrar una solución que no era la óptima(41285).

5) Autómata de aprendizaje: Mustafa Misir, Tony Mauters, Katja Veerbeck y Greet Vanden Berghe[11] proponen en el 2009 usar un autómata de aprendizaje (LA) para resolver TTP.

Un LA es un mecanismo que consiste en un conjunto de acciones a y una distribución de probabilidad asignada a ese conjunto, p, inicialmente uniforme, es decir, cada acción tiene la misma probabilidad de ser escogida.

En resumidas cuentas, si una una acción $a_i \in a$ es aleatoriamente escogida y lleva a resultados positivos, su probabilidad $p_i \in p$ aumenta para futuras iteraciones y si lleva a resultados negativos, disminuye para el futuro.

En este caso, existen 5 "acciones", cada una representando un tipo de permutación aplicable a una solución actual del TTP. Estos 5 tipos de permutaciones son los mismos 5 tipos de movimientos que se usaron en la implementación de SA (ver sección 2.C.2)[1]. Al elegir una acción según su probabilidad, se genera un vecindario desde la solución actual usando permutaciones referentes a la acción. Por ejemplo, si se eligió SwapTeams, se genera soluciones vecinas a partir de aplicar SwapTeams a la solución actual sobre distintos equipos.

Luego de elegir la acción y generar el vecindario, se busca alguna solución vecina que mejore la función objetivo (distancia total recorrida, más penalizaciones por violación de restricciones blandas norepeat y atmost). Entre mejores soluciones se encuentren, más aumentará la probabilidad de escoger ese tipo de permutación en el futuro y si se encuentran muchas soluciones no mejores, dicha probabilidad bajará.

También es posible que las probabilidades se reinicien volviendo a ser equitativas, y así evitar estancarse en los mismos tipos de movimientos.

La motivación detrás del uso de LA para resolver TTP [11] reside en que es mejor que "la máquina decida" que tipo de movimientos (o heurísticas) son los que llevarán a mejores soluciones más rápidamente, especialmente en este tipo de problemas, en donde hay muchos tipos de permutaciones y darle prioridad a solo un grupo de ellas puede no ser la mejor opción en todos los casos.

Para los experimentos, se usaron las instancias de National League de $N = 4$, $N = 6$, $N = 8$, $N = 10$, $N = 12$, $N = 14$ y $N = 16$, las cuales pasaron varias veces por la implementación hecha, mientras con distintos hiperparámetros. En la siguiente tabla se muestran los resultados, tanto en distancia total recorrida de las soluciones finales (mínimo, máximo, promedio y desviación estándar) como en tiempo (tanto en segundos como en número de iteraciones):

LA	NL4	NL6	NL8	NL10	NL12	NL14	NL16
AVG	8276	23916	39802	60046	115828	201256	288113
MIN	**8276**	**23916**	**39721**	**59583**	**112873**	**196058**	**279330**
MAX	8276	23916	40155	60780	117816	206009	293329
STD	0	0	172	335	1313	2779	4267
TIME	~0	0.062	0.265	760	3508	1583	1726
ITER	1.90E+01	1.34E+03	8.23E+04	1.94E+08	6.35E+08	2.14E+08	1.79E+08

Fig. 5. Resultados del autómata de aprendizaje en las instancias de National League

De la misma forma, se probó esta implementación en las instancias del benchmark "super" de $N = 4$, $N = 6$, $N = 8$, $N = 10$, $N = 12$, $N = 14$, obteniendo los siguientes resultados:

LA	Super4	Super6	Super8	Super10	Super12	Super14
AVG	71033	130365	182975	327152	475899	634535
MIN	**63405**	**130365**	**182409**	**318421**	**467267**	**599296**
MAX	88833	130365	184098	342514	485559	646073
STD	12283	0	558	6295	5626	13963
TIME	~0	0.031	1	1731	3422	1610
ITER	8.00E+00	9.41E+02	1.49E+05	3.59E+08	5.12E+08	2.08E+08

Fig. 6. Resultados del autómata de aprendizaje en las instancias Super

En la siguiente tabla, se realiza una comparación entre los resultados obtenidos de esta implementación y los mejores resultados obtenidos hasta esa fecha (2009), (en caso de que estos estuvieran disponibles):

TTP Inst.	LHH	Best	Difference (%)
NL4	**8276**	**8276**	0,00%
NL6	**23916**	**23916**	0,00%
NL8	**39721**	**39721**	0,00%
NL10	59583	59436	0,25%
NL12	112873	110729	1,88%
NL14	196058	188728	3,88%
NL16	279330	261687	6,74%
Super4	**63405**	**63405**	0,00%
Super6	**130365**	**130365**	0,00%
Super8	**182409**	**182409**	0,00%
Super10	318421	316329	0,66%
Super12	**467267**	–	–
Super14	**599296**	–	–

Fig. 7. Comparación entre resultados del autómata de aprendizaje y mejores resultados encontrados hasta esa fecha para instancias National League (NL) y Super

Considerando que los resultados que se obtuvieron en esta implementación alcanzaron o se acercaron bastante a los mejores resultados obtenidos hasta esa fecha, y que los tiempos de ejecución para instancias de valores grandes de N ($N \geq 10$) fueron de pocas horas (considerando que usualmente en esos casos demora días (ver secciones 2.C.1 y 2.C.2)), LA resulta ser un excelente método para resolver TTP, considerando que según los desarrolladores de esa implementación [11], inicialmente no se esperaba que se llegaran a soluciones de tal calidad en tan poco tiempo usando estas heurísticas.

6) Depth First Search: En el año 2009, David C. Uthus, Patricia J. Riddle y Hans W. Guesgen [16] proponen resolver TTP usando un método basado Depth First Search (DFS).

Usualmente se habla de DFS como un algoritmo de recorrido de grafos, pero se puede abstraer a la construcción de soluciones de problemas con cierto nivel de restricciones. Básicamente, en cada fase se genera parte de la solución, según el dominio de decisiones disponibles que se actualiza en cada fase.

Cada decisión tomada en una fase restringe las decisiones que se podrán elegir en las fases futuras (para acelerar el proceso se puede usar forward checking, es decir, actualizar el dominio de decisiones cada vez que se toma una decisión en cada fase, para que en el futuro no haya que estar revisando constantemente si cada decisión cumple con las restricciones). En otras palabras, las restricciones se propagan de fase en fase.

Si en una fase el conjunto de decisiones está vacío, quiere decir que no se puede obtener una solución factible habiendo tomado alguna de las decisiones hechas en fases anteriores. Por lo tanto, se retrocede de fase (backtraking) y se toma otra decisión.

En caso de que se completen todas las fases, se habrá obtenido una solución factible. Lo siguiente es volver a retroceder de fase tomando otras decisiones. El objetivo es encontrar todas las soluciones factibles posibles y de esas escoger la mejor, aunque se le pude poner un tiempo límite al algoritmo, como se realiza en esta implementación (en este caso retornaría la mejor solución encontrada durante ese tiempo). Todo este proceso es comparable a una búsqueda en arboles, donde cada nivel es una fase, cada nodo un estado de la solución y cada eje una decisión.

En este caso, en cada fase se escogen los partidos que se jugarán en cada ronda y que, según los partidos asignados en rondas (fases) anteriores, no generen conflictos en las restricciones. Al finalizar todas las fases (habiendo tomado las decisiones que no generen conflictos), se calcula la distancia total de viajes, se retrocede de fase (ronda) y se toman otras decisiones.

Normalmente este proceso de búsqueda completa debería ser muy ineficiente, puesto que el tamaño del espacio de soluciones que hay que explorar es extremadamente grande, especialmente para valores de N grandes. El tiempo de ejecución para esta implementación es reducido añadiendo algunas modificaciones al algoritmo. Algunas de ellas son:

1) En TTP, la mitad de soluciones factibles posibles es una simetría de la otra mitad. Si una solución B es una simetría de otra solución A, quiere decir que tiene la misma organización de partidos por ronda, pero se invierten los roles local/visita en cada juego; sin embargo, estas soluciones tienen la misma distancia total de viaje. Aprovechando este hecho, la implementación solo explora una de las dos mitades del espacio de soluciones, lo que reduce en un 50% el hipotético tiempo de ejecución.
2) en lugar de recorrer solo un árbol de decisiones, se recorren distintos arboles "subarboles", que se generan al inicio. En cada subarbol, hay una cantidad (máxima de 4) de parejas distintas fijas de equipos que jugarán en la primera ronda. Durante el recorrido de estos arboles, estas parejas no pueden ser cambiadas.
3) continuando con la idea de los subarboles, cada uno de estos es recorrido en paralelo, compartiendo cada thread memoria en referencia a las mejores soluciones encontradas hasta el momento y las distancias óptimas estimadas (ver modificación 1 de esta lista).

La implementación se probó en instancias del benchamrk "super" de $N = 4$, $N = 6$, $N = 8$, $N = 10$, $N = 12$ y de $N = 14$. La siguiente tabla muestra los resultados obtenidos, en tiempo, distancia total de viajes de solución obtenida ("LB")

y distancia total de viajes de solución óptima encontrada hasta esa fecha ("UB"):

Instance	LB	UB	Time
SUPER4	63405	63405	0.0
SUPER6	130365	130365	0.27
SUPER8	182409	182409	361.20
SUPER10	316329	316329	710 236
SUPER12	367812	-	637
SUPER14	467839	-	98 182

Fig. 8. Resultados del autómata de aprendizaje en las instancias de National League

Como se ve ve en el gráfico, para valores de N desde 4 hasta 10 se llegó a la solución óptima (para $N = 12$ t $N = 14$ no se ha encontrado una solución óptima hasta esa fecha). Los tiempos de búsqueda son bastante bajos, especialmente en $N = 12$ y $N = 14$, en que normalmente se tardaría muchos días.

3. Técnica Propuesta - Simmulated Annealing

A continuación, se propone un algoritmo basado en Simmulated Annealing o SA para resolver TTP, inspirado mayormente por las implementaciones de esta misma heurística mencionadas (ver sección 2.C.2[1] y 2.C.3[12]).

Una vez más, SA es un método de búsqueda de soluciones categorizado como un método reparador, es decir, realiza permutaciones o transformaciones a una solución ya generada para mejorarla. SA se distingue por poder aceptar permutaciones que lleven a soluciones no mejores según una función de evaluación, con una probabilidad dependiente de una variable "temperatura" y de la función de evaluación mencionada.

En esta sección se realizará una descripción del algoritmo propuesto y sus componentes.

A. Representación

La representación de soluciones para esta implementación será la misma que la propuesta por [1], es decir, una tabla S de (N x $2(N-1)$) donde cada fila representa un equipo y cada columna una ronda. El valor que la casilla de la fila i en la columna j, $S_{i,j}$, indica contra qué equipo k debe enfrentarse i en la ronda j.

Si $S_{i,j} = k > 0$, el equipo i jugará de local en la ronda j contra el equipo k. Si $S_{i,j} = -k < 0$, el equipo i jugará de visita en la ronda j contra el equipo k.

B. Soluciones iniciales

Esta implementación tomará como soluciones iniciales torneos DRR generados por el algoritmo circular de Kirkman [9]. Este consiste en generar varios emparejamientos de equipos (de $N/2$ partidos en cada emparejamiento) tales que al insertarlos uno en cada ronda, se genere un torneo RR factible.

El primer paso de este algoritmo es generar aleatoriamente un emparejamiento inicial E_0, como en la siguiente figura: (suponiendo un caso $N = 6$)

Fig. 9. Ejemplo de E_0. En este caso, los equipos de arriba jugarían de local y los de abajo de visita. El equipo 1 (L) juega contra el 4 (V), el 2 (L) contra el 5 (V) y el 3 (L) contra el 6(V)

Para generar más emparejamientos, a partir del inicial, se aplica la función *rotación(E_0,p)*, la cual "fija" uno de los equipos de E_0 (de manera determinista) y "rota" el resto de equipos p veces ($p \in [0, N-2]$). Siguiendo con el ejemplo anterior, así se verían los emparejamientos al rotar E_0 por cada valor posible de p:

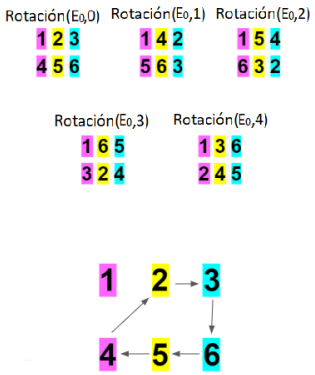

Fig. 10. Ejemplo de rotaciones posibles de E_0, según distintos valores de p. La forma en que se rotan los equipos de describe abajo. Nótese que: rotación$(E_0, 0) = E_0$

Si cada uno de estos emparejamientos se inserta en una ronda, se generaría un RR factible. Para crear un DRR, se toman en consideración para cada emparejamiento, su inverso o "-rotación(E_0, p)":

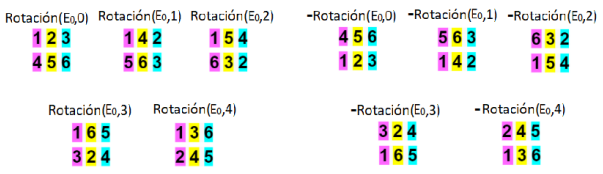

Fig. 11. Ejemplo de rotaciones posibles de E_0 (a la izquierda), y sus respectivos inversos (a la derecha), en los cuales se invierten los roles de local (tríos de "arriba") y visita (tríos de "abajo")

Si se llenan las $2 * (N-1)$ rondas con estos emparejamientos (asignando aleatoriamente rondas con emparejamientos), queda una solución de DRR factible.

El pseudocódigo de este proceso es:

procedure INITIAL_DRR
  $E_0 \leftarrow$ emparejamiento aleatorio inicial de los N equipos.
  $E \leftarrow \emptyset$
  $DRR \leftarrow$ torneo inicialmente vacío.
  for p in $[0, N-2]$ **do**
    $E \leftarrow E \cup$ rotación(E_0, p)
    $E \leftarrow E \cup$ -rotación(E_0, p)
  end for
  ordenar aleatoriamente E
  for r in $range(2(N-1))$ **do**
    insertar juegos de $E[r]$ en ronda r de DRR.
  end for
  retornar DRR
end procedure

Es necesario dejar en claro que este algoritmo crea una instancia de DRR, la cual no necesariamente cumple con las restricciones propias de una solución de TTP. En otras palabras, este algoritmo no garantiza que la solución inicial cumpla con las restricciones atmost y norepeat. Es más, dada la naturaleza combinatoria de este problema y a la aleatoriedad del algoritmo circular, es casi seguro que la solución inicial presente violaciones a estas restricciones, volviéndola infactible.

Esta infactibilidad se debería solucionar en las otras etapas de la implementación propuesta, aunque al igual que en las implementaciones de [1] y [12], se explorará en el espacio de soluciones infactibles, al relajar las restricciones atmost y no repeat. Por otro lado, se penalizará durante la búsqueda las violaciones a estas restricciones (de una forma que se explicará más adelante). Finalmente, hay que aclarar que la solución final que entregue el algoritmo debe ser factible.

C. mejores soluciones y función de evaluación

En esta implementación se buscarán dos "mejores soluciones" según ámbitos diferentes:

- **La mejor solución factible**, es decir, la mejor solución que cumple con todas las restricciones de TTP (incluyendo atmost y norepeat) y que al mismo tiempo minimiza la función objetivo: la distancia total recorrida.
- **La mejor solución según la función de evaluación**, la cual de ahora en adelante se le llamará simplemente la **"mejor solución actual"**. Esta solución es la que minimiza la siguiente función de evaluación o función costo [12]:

$$c(S) = \sqrt{d(S) + (w(1 + \frac{\sqrt{v(S)}\ln(v(S))}{2})^2} \qquad (1)$$

Donde $d(S)$ es la distancia total recorrida de la solución S (la función objetivo), $v(S)$ es el número de infracciones a las restricciones atmost y norepeat presentes en S (si $V(S) = 0$, la solución es factible) y $w > 1$ es una variable "peso", cuyo propósito será explicado más adelante.

D. Permutaciones

En cada "iteración" del algoritmo, se generará una solución candidata a partir de una permutación a la solución actual. Para esta implementación, se usarán 2 de los 5 tipos de permutaciones explicados en la sección 2.C.2 [1]: $partialSwapRounds(T_i, r_k, r_l)$ y $partialSwapTeams(r_k, T_i, T_j)$. Se escogieron estos dos movimientos debido a permiten explorar una mayor porción del espacio de búsqueda en menor tiempo [12].

E. Iteraciones

En cada iteración se realiza una permutación aleatoria a la solución actual S, generando una solución candidata S'. Para esto, se escoge aleatoriamente a uno de los dos tipos de permutaciones y luego se escogen aleatoriamente los parámetros de los movimientos. Es decir, si se escogió $partialSwapRounds$ se eligen T_i, r_k y r_l ($r_k \neq r_l$), y si se escogió $partialSwapTeams$ se eligen r_k, T_i, T_j ($T_i \neq T_j$).

Antes de verificar si esta solución candidata puede o no ser aceptada como la nueva solución actual ($S \leftarrow S'$), se verifica si puede ser aceptada como la mejor solución **factible**. Es decir, si $v(S') = 0$ y $d(S')$ es la menor distancia encontrada hasta ese momento, para soluciones factibles.

Luego de esto, se calcula la función costo $c(S')$. Se aceptará S' como nueva solución actual si $c(S') < c(S)$. En caso contrario, se puede aceptar también S' con una probabilidad de $e^{\frac{C(S)-C(S')}{T}}$, donde T es la variable temperatura.

Finalmente, se reduce el valor de T levemente, multiplicándolo por $0 < coolRatio \leq 1$, un parámetro del algoritmo. Para evitar que el sistema se "enfríe" en un periodo demasiado corto, es recomendable que el valor de $coolRatio$ sea cercano a 1, pero no igual.

F. Oscilación estratégica

Si durante una iteración se acepta una solución cuyo costo ($c(S)$) es menor al costo de la mejor solución actual (cuyo valor es inicialmente infinito), esa nueva solución pasa a ser la mejor solución actual.

Cada vez que se encuentre una nueva mejor solución actual, se modifica el valor de la variable w o "peso", presente en la función costo. El propósito de esta variable, como se puede adivinar viendo la función costo, es representar el nivel de gravedad en el que se penalizan las violaciones a las restricciones atmost y norepeat durante la evaluación de una solución candidata, de acuerdo con la función de evaluación.

Si la nueva mejor solución actual no es totalmente factible ($v(S) > 0$), el valor de w aumenta, al multiplicarse por un

hiperparámetro $ratioW > 1$. Si en cambio, la nueva mejor solución actual es factible ($v(S) = 0$), el valor de w disminuye, al multiplicarse por el mismo hiperparámetro $ratioW$.

Este proceso se llama "oscilación estratégica", y fue utilizado en [1] con el fin de variar la "urgencia" de encontrar soluciones factibles. Si la mejor solución actual no es factible, esta urgencia aumenta con w y, en el caso contrario, disminuye con w la urgencia de encontrar soluciones factibles.

G. Recalentamientos y ciclos

A medida que ocurran muchas iteraciones, el valor de T disminuirá de tal forma que será cada vez más difícil aceptar una solución no mejor según la función de evaluación, lo que podría acabar en un estancamiento de óptimos locales. Para solucionar esto, se emplean los recalentamientos.

Dado un parámetro $maxC$, cada vez que se ejecuten $maxC$ iteraciones consecutivas en las que no se acepte una solución candidata (esto sucederá cuando la solución actual no tenga en su vecindario muchas soluciones mejores que que esta y cuando el valor de T sea lo suficientemente bajo como para no aceptar ninguna solución no mejor), se realizará un recalentamiento. Es decir, se reiniciará T a su valor inicial T_0, siendo T_0 otro parámetro del algoritmo.

Dado otro parámetro $maxR$, una vez que se realicen $maxR$ recalentamientos empezará un nuevo "ciclo", es decir, se trabajará desde una nueva solución inicial generada por el algoritmo ya explicado en esta sección. Al comenzar un nuevo ciclo, además, se reinicia la mejor solución actual.

H. Fin del algortimo

Existe un parámetro llamado "$maxIterations$". Tal como su nombre deja intuir, este parámetro indica el número máximo de iteraciones totales que el algoritmo ejecutará. Una vez este número se traspase, el algoritmo devuelve la mejor solución factible detectada durante toda la ejecución. En caso de no haber detectado ninguna solución factible en $maxIterations$ iteraciones, el algoritmo habrá fallado. Por lo tanto, hay que ser cuidadoso al momento de definir el valor de este parámetro.

I. Parámetros

En total, estos son los parámetros que recibirá el algoritmo, a parte de la matriz distancia NxN:

- $T_0 > 1$: temperatura inicial.
- $0 < coolRatio < 1$: ratio de enfriamiento. Indica la reducción de la temperatura en cada iteración.
- $w_0 > 0$: peso inicial de penalización de violación a restricciónes.
- $ratioW \geq 1$: ratio de cambio de peso. Indica que tanto aumenta o se reduce w al encontrar una mejor solución actual.
- $maxC > 1$: cantidad de iteraciones consecutivas en las que no se acepta una solución candidata que tienen que suceder para que se realice un recalentamiento.
- $maxR > 1$: cantidad de recalentamientos que tienen que suceder para cambiar de ciclo.
- $maxIterations \geq 1$ numero de iteraciones que el algoritmo realizará.

J. Pseudocódigo

El pseudocódigo de todo el proceso es el siguiente:

```
procedure TTP_SIMULATED_ANNEALING
    factibleFound ← false
    bestFactibleDist ← ∞
    T ← T_0
    r ← 0
    i ← 0 # iteraciones
    w = w_0
    while r < R and i < maxIterations do
        generar solución inicial S usando método circular
        bestS ← S
        bestCost ← C(S)
        q ← 0
        while q < Q and i < maxIterations do
            c ← 0
            while c < C and i < maxIterations do
                S' ← permutacionAleatoria(S)
                if v(S') == 0 and d(S') < bestFactibleDist
then
                    bestFactibleDist ← d(S')
                    bestFactibleS ← S'
                    factibleFound ← True
                end if
                eval ← e^{-(C(S')-C(S))/T}
                rand ← random([0, 1])
                if C(S') < C(S) or eval < rand then
                    c ← 0
                    S ← S'
                    if C(S) < bestCost then
                        bestCost ← C(S)
                        if v(S) == 0 then w ← w * ratioW
                        end if
                    else
                        w ← w/ratioW
                    end if
                else
                    c ← c + 1
                end if
                T ← T * coolRatio
            end while
            T ← T_0
            q ← q + 1
        end while
        r ← r + 1
    end while
    if factibleFound then return bestFactibleS
    else return false #algoritmo falló
    end if
end procedure
```

4. Algoritmo Greedy previamente propuesto

Previo a la realización de este informe, se propuso en un informe anterior un algoritmo basado en greedy search para resolver TTP [3]. Antes de ver los resultados de la implementación de SA hecha en este informe (ver sección 3.B), se describirá brevemente el algoritmo greedy mencionado.

Desde un punto de partida, obtenido utilizando el mismo algoritmo circular [9] que se utilizó para crear soluciones iniciales en la implementación de SA propuesta en este informe, se elige en varias iteraciones la permutación que minimice $v(S)$, o si hay un empate entre permutaciones, la que minimice $d(S)$ hasta llegar a una solución factible ($v(S) = 0$).

Las permutaciones permitidas son dos de las mencionadas por [1], esta vez $swapRounds$ y $swapTeams$.

Esta implementación greedy se probó en varias instancias del benchmark NL. Para cada instancia, se obtuvieron 1000 soluciones factibles distintas utilizando este método. En la siguiente tabla, se muestran los resultados según función objetivo F (el menor valor encontrado, el mayor encontrado y el promedio de valores) y tiempo total de ejecución (de las 1000 repeticiones).

Instancia	$\frac{Sols.Factibles}{Sols.Totales}$	F_{min}	F_{max}	$\overline{F}$	Tiempo[S]
NL4	1000/1000	8559	11594	9870	0.05
NL6	1000/1000	26256	35875	30639	0.55
NL10	1000/1000	79454	98500	88645	11.28
NL12	1000/1000	152329	183735	165987	24.4632
NL14	1000/1000	286093	347414	316686	52.94
NL16	1000/1000	409915	494346	454763	104.216

5. Escenario experimental

Se probará una implementación del algoritmo de SA propuesto en este informe, echo en C++, desde un sistema operativo Ubuntu 20.04, corriendo con un procesador intel core i3-6006 CPU @ 2.00Hz.

En esta implementación, los valores de algunos de los parámetros están definidos según el valor de N. Estos parámetros son:

- $w_0 = 500 * N$
- $ratioW = 1.2$
- $maxC = 50 * N$

Por otra parte los valores de T_0, $coolRatio$, $maxIterations$ y $maxR$ son definidos por la persona que ejecute el algoritmo, quien debe ingresarlos por consola.

Cada uno de los experimentos se realizará en las instancias $NL10$, $NL12$, $NL14$ y $NL16$ del benchmark National League.

En cada experimento se probará el algoritmo en distintas configuraciones de parámetros. Cada configuración será probada en 4 semillas.

Luego de cada "prueba" se medirá el tiempo de ejecución y la función objetivo de la solución obtenida. De ambas métricas se obtendrán los mínimos (min), máximos (max), medias ($mean$) y desviaciones estándar (std) de las 4 ejecuciones hechas para configuración de parámetros.

A. Experimento 1: numero de iteraciones

En este experimento, se ejecutará el algoritmo en cada instancia con distintos valores de $maxIterations$: 100,000, 500,000 y 1,000,000 de iteraciones.

Durante todo el experimento 1, los valores de T_0, $coolRate$ y $maxR$ serán iguales a 5000, 0.9999 y 7, respectivamente.

B. Variación de temperatura

En este experimento, se ejecutará el algoritmo en cada instancia para distintos valores de T_0 y $coolRatio$. Dado que ambos parámetros están directamente relacionados con la temperatura del sistema durante la ejecución, se probará con las siguientes combinaciones de valores de estos parámetros:

- $T_0 = 5,000$, $coolRatio = 0.9999$
- $T_0 = 10,000$, $coolRatio = 0.9999$
- $T_0 = 5,000$, $coolRatio = 0.9995$
- $T_0 = 10,000$, $coolRatio = 0.9995$

Durante todo el experimento 2, el valor de $maxIterations$ será igual a $500,000$ y el valor de $maxR$ será igual a 7.

C. Experimento 3: máximo de recalentamientos

Para este último experimento, se probará la implementación para cada instancia, variando los valores del parámetro $maxR$. Esta vez, se usarán dos valores:

- $maxR = 7$
- $maxR = 14$

Durante todo este experimento, los valores de T_0, $coolRate$ y $maxIterations$ serán iguales a 5000, 0.9999 y $500,000$, respectivamente.

6. Resultados obtenidos

A. Experimento 1

Para el experimento 1, se obtuvieron los siguientes resultados:

Para la instancia $NL10$:

		min	max	$mean$	std
$maxIterations = 100000$	Tiempo (S)	1.46	1.51	1.49	0.019
	Distancia	72246	74068	72739	769.74
$maxIterations = 500000$	Tiempo (S)	7.07	7.20	7.13	0.019
	Distancia	69978	70743	70481	769.74
$maxIterations = 1000000$	Tiempo (S)	14.03	14.15	14.08	0.043
	Distancia	69978	70743	70410	340.45

Para la instancia $NL12$:

		min	max	$mean$	std
$maxIterations = 100000$	Tiempo (S)	1.96	2.03	1.98	0.029
	Distancia	138462	139923	139386	602.83
$maxIterations = 500000$	Tiempo (S)	9.56	9.62	9.60	0.023
	Distancia	132459	139925	136652	3337.2
$maxIterations = 1000000$	Tiempo (S)	19.14	19.45	19.22	0.131
	Distancia	132073	133471	132520	571.39

Para la instancia $NL14$:

		min	max	$mean$	std
$maxIterations = 100000$	Tiempo (S)	2.53	2.57	2.55	0.011
	Distancia	248600	258601	252570	4228.81
$maxIterations = 500000$	Tiempo (S)	12.58	12.80	12.66	0.088
	Distancia	246096	258601	250474	4802.5
$maxIterations = 1000000$	Tiempo (S)	25.10	25.34	25.18	0.092
	Distancia	242776	247386	245911	1885.05

Para la instancia $NL16$:

		min	max	$mean$	std
$maxIterations$ = 100000	Tiempo (S)	3.13	3.20	3.17	0.03
	Distancia	366841	369410	368001	1173.29
$maxIterations$ = 500000	Tiempo (S)	15.60	15.68	15.63	0.03
	Distancia	366316	361628	363290	1921.51
$maxIterations$ = 1000000	Tiempo (S)	31.29	31.37	31.33	0.03
	Distancia	356029	363481	360692	2796.17

En todas las ejecuciones se logró llegar a una solución factible, dentro del límite de iteraciones.

B. Experimento 2

Para el experimento 2, se obtuvieron los siguientes resultados:

Para la instancia $NL10$:

		min	max	$mean$	std
T_0 = 5000	Tiempo (S)	7.05	7.21	7.11	0.06
$coolRatio$ = 0.9999	Distancia	69978	70743	70481	312.56
T_0 = 10000	Tiempo (S)	7.04	7.29	7.14	0.09
$coolRatio$ = 0.9999	Distancia	70385	72323	71468	694.52
T_0 = 5000	Tiempo (S)	7.00	7.07—	7.02	0.03
$coolRatio$ = 0.9995	Distancia	68688	69432	68892	313.36
T_0 = 10000	Tiempo (S)	7.05	7.11	7.08	0.03
$coolRatio$ = 0.9995	Distancia	69273	70967	70354	696.73

Para la instancia $NL12$:

		min	max	$mean$	std
T_0 = 5000	Tiempo (S)	9.57	9.68	9.61	0.05
$coolRatio$ = 0.9999	Distancia	132459	133925	133652	705.27
T_0 = 10000	Tiempo (S)	9.64	9.77	9.71	0.05
$coolRatio$ = 0.9999	Distancia	134753	137072	136084	1017.33
T_0 = 5000	Tiempo (S)	9.57	9.63	9.61	0.02
$coolRatio$ = 0.9995	Distancia	133109	134598	133610	607.92
T_0 = 10000	Tiempo (S)	9.54	9.66	9.58	0.05
$coolRatio$ = 0.9995	Distancia	131116	135161	132267	1674.87

Para la instancia $NL14$:

		min	max	$mean$	std
T_0 = 5000	Tiempo (S)	12.57	12.80	12.65	0.09
$coolRatio$ = 0.9999	Distancia	246095	258601	250474	4802.28
T_0 = 10000	Tiempo (S)	12.59	12.68	12.63	0.03
$coolRatio$ = 0.9999	Distancia	241153	252208	247182	5075.60
T_0 = 5000	Tiempo (S)	12.59	12.86	12.68	0.11
$coolRatio$ = 0.9995	Distancia	243433	254758	250197	4152.67
T_0 = 10000	Tiempo (S)	12.55	12.61	12.57	0.20
$coolRatio$ = 0.9995	Distancia	243236	250951	246363	3293.23

Para la instancia $NL16$:

		min	max	$mean$	std
T_0 = 5000	Tiempo (S)	15.62	15.68	15.63	0.03
$coolRatio$ = 0.9999	Distancia	361628	366316	363290	1921.51
T_0 = 10000	Tiempo (S)	15.58	15.68	15.64	0.03
$coolRatio$ = 0.9999	Distancia	355639	364252	358422	3519.53
T_0 = 5000	Tiempo (S)	15.46	15.62	15.52	0.06
$coolRatio$ = 0.9995	Distancia	352019	362261	355538	4185.19
T_0 = 10000	Tiempo (S)	15.50	15.78	15.58	0.13
$coolRatio$ = 0.9995	Distancia	352142	357661	354266	2052.22

En todas las ejecuciones se logró llegar a una solución factible, dentro del límite de iteraciones.

C. Experimento 3

Para el experimento 3, se obtuvieron los siguientes resultados:

Para la instancia $NL10$:

		min	max	$mean$	std
$maxR$ = 7	Tiempo (S)	7.07	7.20	7.12	0.019
	Distancia	69978	70743	70481	769.74
$maxR$ = 14	Tiempo (S)	7.08	7.31	7.18	0.08
	Distancia	71066	72231	71758	497.71

Para la instancia $NL12$:

		min	max	$mean$	std
$maxR$ = 7	Tiempo (S)	9.56	9.62	9.6	0.023
	Distancia	132459	139925	136652	3337.2
$maxR$ = 14	Tiempo (S)	9.57	9.65	9.61	0.03
	Distancia	135293	136486	136173	508.31

Para la instancia $NL14$:

		min	max	$mean$	std
$maxR$ = 7	Tiempo (S)	12.58	12.80	12.66	0.088
	Distancia	246096	258601	250474	4802.5
$maxR$ = 14	Tiempo (S)	12.52	12.75	12.61	0.08
	Distancia	248541	253813	249889	2265.94

Para la instancia $NL16$:

		min	max	$mean$	std
$maxR$ = 7	Tiempo (S)	15.60	15.68	15.63	0.03
	Distancia	366316	361628	363290	1921.51
$maxR$ = 14	Tiempo (S)	15.56	15.89	15.71	0.13
	Distancia	356580	360564	357887	1626.89

En todas las ejecuciones se logró llegar a una solución factible, dentro del límite de iteraciones.

(Nota: Para los tres experimentos, las distancias mínimas y los tiempos de ejecución mínimos presentes en cada fila no necesariamente pertenecen a la misma ejecución. Lo mismo con las distancias máximas y tiempos máximos.)

D. Análisis de los resultados

En términos generales, el hecho de que el algoritmo logró llegar a una solución factible dentro del límite de iteraciones en todas las ejecuciones de todos los experimentos indica que los valores de los parámetros establecidos en todo el escenario experimental fueron más que suficientes para alcanzar la factibilidad.

Al comparar estos resultados con los obtenidos en la implementación de Greedy (sección 4), se observa que incluso las soluciones menos mejores obtenidas de SA tienen menores distancias que las mejores soluciones obtenidas por greedy. Aunque los tiempos de ejecución de greedy son muchos menores, más aún cuando los mostrados en la sección 4 corresponden cada uno a 1000 repeticiones del algoritmo, SA fue fue capaz de encontrar soluciones mucho mejores en tiempos razonables (en menos de un minuto). Por lo tanto, es correcto afirmar que el algoritmo de SA es mejor que el de greedy.

Sin embargo, si comparamos los resultados con los obtenidos por otras implementaciones discutidas en este informe como por ejemplo el autómata de aprendizaje [11] (sección 2.C.5), se observa que para las instancias probadas en la implementación de SA, se encontraron soluciones de distancias mucho menores, aunque en un mayor tiempo (según la Fig.5, más de 10 minutos mientras que en el SA implementado nunca más de un minuto).

Asimismo, las otras implementaciones de SA expuestas en este informe, propuestas por [1] (sección 2.C.2) y [12] (sección 2.C.3) lograron alcanzar soluciones de mucho menor distancia pero en un tiempo mucho mayor, desde poco más que diez minutos hasta varios días (aunque de las instancias probadas en esta versión de SA, la implementación de [12] solo se probó en NL10). Dada que gran parte de la implementación propuesta en este informe se inspiró en estas dos propuestas, se podría teorizar que ésta sería capaz de alcanzar soluciones con menor distancia si se le diera un tiempo máximo de búsqueda mucho mayor (es decir, aumentando los valores de $maxIterations$, $maxR$ y $maxC$).

Comparando las desviaciones estándar con los promedios, los promedios tienen ordenes de magnitud 2 o 3 veces mayores que las desviaciones estándar. Esto quiere decir que los resultados, tanto en distancia como en tiempo, son muy poco variados entre varias ejecuciones con los mismos parámetros, pero con semillas distintas. Se podría contra argumentar que el número de semillas que se probó para cada configuración de parámetros en cada instancia (4 semillas) es muy bajo para sacar una conclusión como esta, pero este patrón se repite para todas las instancias, en todas las configuraciones de parámetros. Por eso es difícil que esto sea una simple coincidencia.

Observando los resultados del experimento 1, es fácil darse cuenta que existe una relación directa entre número de iteraciones y tiempo de ejecución. Esto parece ser obvio puesto que casi todo el tiempo invertido durante la ejecución del algoritmo se invierte en las iteraciones. Es más, la relación de tiempo y máximo de iteraciones es claramente lineal, puesto que entre los resultados de $maxIterations = 100000$ y $maxIterations = 500000$ los promedios de tiempo se multiplican por aproximadamente 5 y entre $maxIterations = 500000$ y $maxIterations = 1000000$ los tiempos promedio se multiplican por aproximadamente 2.

Asimismo, el tiempo también aumenta un poco al aumentar el valor de N, variando las instancias. Esto sucede porque los tiempos que demora el algoritmo en realizar las permutaciones y evaluar las soluciones candidatas durante cada iteración dependen del valor de N.

En experimento 2, según los promedios, para $NL10$ se obtuvieron mejores soluciones en cuanto a distancia cuando los valores de T_0 y $coolRatio$ fueron bajos; para las instancias $NL12$, $NL14$ y $NL16$ en cambio las mejores soluciones se obtuvieron cuando el valor de T_0 fue alto y $coolRatio$ fue bajo. Sin embargo en este caso, las desviaciones estándar fueron los suficientemente grandes para indicar que se pueden obtener soluciones mejores con todas las configuraciones de parámetros disponibles en este experimento.

Todo parece apuntar a que la mejor configuración de parámetros T_0 y $coolRatio$ depende de la situación y si se quisiera descifrar con mayor exactitud, se debería probar con más semillas y con mayor tiempo de búsqueda.

Asimismo, los tiempos de ejecución para cada instancia no varían demasiado al modificar los parámetros T_0 y $coolRatio$, lo que indica que estos no generan un impacto muy significativo en el tiempo de ejecución.

En cuanto el experimento 3, todo indica que un mayor valor de $maxR$ llevará a mejores soluciones. Esto parece obvio, debido a que $maxR$ está relacionado al tiempo que se le da al algoritmo para explorar el espacio de búsqueda desde una misma solución inicial. Por lo tanto, cuando el valor de $maxR$ es alto, se explota más el potencial de cada solución inicial para llevar al algoritmo a una mejor solución.

Contrario a lo que se podría haber adivinado, los tiempo de ejecución no parecen aumentar demasiado con valores altos de $maxR$. Es más, para la instancia $NL14$, el tiempo promedio de $maxR$ son menor cuando el valor de $maxR$ es alto. Por lo tanto, según estos resultados $maxR$ no es un parámetro que impacte significativamente en el tiempo de ejecución.

Una hipótesis de porqué ocurre esto último es que cuando se explora el espacio de búsqueda desde una solución inicial por suficientes iteraciones, es posible estancarse en óptimos locales de tal forma que no se acepte ninguna solución candidata (aún con la diversificación que ofrece SA), lo que provoca que la cantidad de soluciones rechazadas de forma consecutiva aumente rápidamente hasta sobrepasar $maxC$, haciendo que los ciclos duren mucho menos (ver sección 3.G) y, en consecuencia, que el tiempo de ejecución no aumente demasiado.

7. Conclusiones

Simmulated annealing es una heurística que se destaca por ofrecer una búsqueda de soluciones fuerte tanto en intensificación como en diversificación. A partir de esta implementación se demostró que con SA se pueden llegar a soluciones bastante decentes (en comparación con el método greedy previamente propuesto) y en tiempos no muy grandes (en menos de un minuto).

Sin embargo, las otras implementaciones de SA expuestas en este informe indican que para garantizar mejores resultados es muy necesario permitirle al algoritmo tiempos de ejecución mucho mayores a los obtenidos por esta implementación, los cuales se limitaron al probar el algoritmo con valores no muy altos de $maxIterations$, $maxR$ y $maxC$ a favor de un proceso más rápido de experimentación.

A modo de conclusión, SA es un muy buen método de búsqueda, pero es muy difícil que encuentre soluciones globalmente óptimas si no existe la disposición de invertir un tiempo de ejecución demasiado alto (llegando a incluso días). Analizando también los resultados de los otros métodos expuestos en este informe, se concluye que TTP es un problema extremadamente complejo y por lo tanto, es difícil lograr

un acercamiento tanto eficaz como eficiente sin recurrir a métodos “poco tradicionales”, como la programación paralela, por ejemplo.

REFERENCES

[1] L. Van Hentenryck P. Vergados Y Anagnostopoulos, A. Michel. A simulated annealing approach to the traveling tournament problem. 2003.

[2] Min Kim B. Iterated local search for the traveling tournament problem. 2012.

[3] Y. Berant. Travelling tournament problem: unacercamiento greedy. 2020.

[4] D. de Werra. Some models of graphs for scheduling sports competitions. *Discrete Applied Mathematics*, 21(1):47–65, 1988.

[5] futbolargentino. recaudacion record para la final de la uefa champions league, url: ”https://www.futbolargentino.com/liga-de-campeones/noticias/recaudacion-record-para-la-final-de-la-uefa-champions-league-230053”.

[6] John H. Holland. Genetic algorithms and adaptation. *NATOCS*, 16(1):317–333, 1984.

[7] Michael A. Trick Kelly Easton, George Nemhauser. The traveling tournament problem description and benchmarks. *Carnegie Mellon University Research Showcase*, 2001.

[8] Michael A. Trick Kelly Easton, George Nemhauser. Solving the traveling tournament problem: A combined integer programming and constraint programming approach. *Lecture Notes in Computer Science*, 2002.

[9] Reverend Kirkman. On a problem in combinatorics. camb. *Dublin Math*, J. 2,:191–204, 1984.

[10] R. Lewis and J Thompson. On the application of graph colouring techniques in round-robin sports scheduling. 2011.

[11] Verbeeck. K Vanden Berghe. G Misir. M, Wauters. T. A new learning hyper-heuristic for the traveling tournament problem. The VIII Metaheuristics International Conference, 2009.

[12] Eshghi K. Nourollahi, S. and H Shokri Razaghi. An efficient simulated annealing approach to the travelling tournament problem. *Computers and Operations Research*, 38(1):190–204, 2012.

[13] Gelatt Jr. M P. Vecchi S, Kirkpatrick. C D. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[14] J. Thakare, L. Umale and B. Thakare. Solution to traveling tournament problem using genetic algorithm on hadoop. International Conference on Emerging Trends in Electrical, Electronics and Communication Technologies-ICECIT, 2012.

[15] TUDN. ”los juegos olimpicos rio 2016 costaron 13 mil millones de dolares”, url: ”https://www.futbolargentino.com/liga-de-campeones/noticias/recaudacion-record-para-la-final-de-la-uefa-champions-league-230053”.

[16] Guesgen. H Uthus. D, Riddle. J. Dfs* and the travelling tournament problem. International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, 2009.