Un acercamiento Evolutivo para Travelling Tournament Problem

Yoel Berant Elorza, Universidad Técnica Federico Santa María

***Abstract*—En este informe se propone un algoritmo evolutivo no genético para resolver TTP, un problema de otimización que consiste en organizar un torneo entre N equipos que cumpla con restricciones y minimice las distancias recorridas por sus equipos.**

Al comparar los resultados obtenidos de este algoritmo con los de dos implementaciones hechas anteriormente, una basada en greedy search y otra basada en simmulated annealing, se obtuvo que el desempeño del algoritmo propuesto no fue tan alto en relación a los de las implementaciones anteriores.

1. Introducción

El "travelling tournament problem" (o TTP) [11] 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 basado en algoritmos evolutivos para entregar una solución al problema. Finalmente, se comparan los resultados con los entregados por otras implementaciones hechas previamente, del tipo Greedy Search y Simmulated Annealing.

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 [8].

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 [22].

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. [11] 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 [11], 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 [11] propusieron un método para resolverlo basado en constraint programming, integer programming y branch and bound [12].

La motivación detrás de esta decisión en cuanto a paradigmas [12] 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 [20] 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" [11], 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 [18] 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[14].

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.

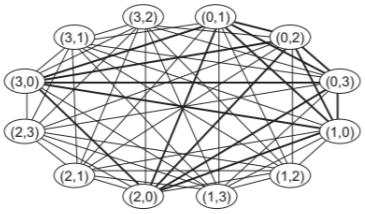


Fig. 3: representación de juegos de un DRR a través de un grafo, con N=4

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*[14], 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.

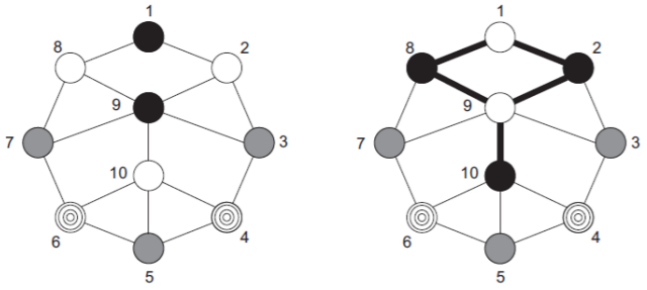


Fig. 4: ejemplo de intercambio de colores en cadena kempe[18].

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 [18].

Para probar esta implementación se usaron instancias del benchmark "Major League Baseball" [6]. 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 [18] alcanza mejores soluciones en un tiempo menor a las demás.

4) Algoritmos genéticos con Hadoop: Laxmi Thakare, Dr. Jayant Umale y Bhushan Thakare [21] proponen en el año 2012 una implementación basada en algoritmos genéticos (GA).

Los GA [10] 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 involu-

cran 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 [21], 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" [11] 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[16] 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 [16] 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 [16], 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 [23] 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. Algoitmos propuestos previamente por el autor de este informe

Antes de avanzar a la propuesta del algoritmo evolutivo, se presentarán también propuestas de otras heurísticas hechas e implementadas por el autor de este informe, así como sus resultados.

A. Greedy Search

Este algoritmo[4] se basa de en el método de búsqueda greedy. La representación de soluciones es la misma representación matricial usada por [1] y [18]. Los pasos que este algoritmo ejecuta son los siguientes:

1) Generar un torneo double round robin. Para eso, usa el método circular de kirkman [13], el cual se explicará más adelante en la sección de este informe 4.
2) Aplicar al torneo el movimiento swapRounds o swapTeams (explicados en la sección 2.C.2) que minimice la cantidad de violaciones a las restricciones atmost y norepeat presentes en el torneo. En caso de que hayan múltiples movimientos que lleven a torneos con la misma mínima cantidad de violaciones, aplicar de esos movimientos el que lleve a un torneo de menor distancia total recorrida.
3) Volver al paso 2, hasta que el torneo no tenga violaciones a la restricciones atmost y norepeat.
4) Retornar torneo

Se probó este algoritmo en las instancias National league para $N = 4$, $N = 6$, $N = 10$, $N = 12$, $N = 14$ y $N = 16$. En cada ejecución, se obtuvieron 1000 soluciones generadas por este algoritmo. A continuación se presentan los resultados de cada ejecución, tanto en promedio de las distancias de las 1000 soluciones generadas ($\overline{F}$), mínima distancia (F_min), máxima distancia (F_{max}) y tiempo de ejecución.

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

B. Simmulated Anneling

Se propuso también otro algoritmo basado en Simmulated Annealing [3], inspirado en las implementaciones de mayormente en las implementaciones de esta misma heurística mencionadas en este informe (ver sección 2.C.2[1] y 2.C.3[18]).

A partir de una solución inicial generada usando el método circular de kirkman[13], el cual será explicado en la sección 4 de este informe, el algoritmo aplicaba permutaciones a las soluciones.

Las permutaciones correspondían a 2 de los 5 tipos de movimientos propuestos por [1]: $partialSwapRounds$ y $partialSwapTeams$. En cada iteración, se escoge uno de estos dos movimientos aleatoriamente, además de las variantes involucradas (T_i,r_k y r_l en el caso de $partialSwapTeams$; y r_k, T_i, y T_j en el caso de $parialSwapRounds$). Como todo algoritmo basado en Simmulated Annealing, además de aceptar movimientos que lleven a mejores soluciones se pueden aceptar movimientos que lleven a soluciones no mejores según una variable probabilística dependiente de una ”temperatura”, que disminuye en cada iteración.

Algo diferenciador de esta implementación, en comparación con las otras implementaciones de SA expuestas en este algoritmo, es el criterio de recalentamientos. Dado un parámetro $maxC > 0$, si no se aceptan $maxC$ permutaciones consecutivas ocurre un recalentamiento, reiniciándose la temperatura a su valor inicial. Además, cada vez que ocurren $maxR$ recalentamientos, se exploran el espacio de búsqueda desde una nueva solución inicial generada aleatoriamente.

Se probó este algoritmo en las instancias de National league de $N = 10$, $N = 12$, $N = 14$ y $N = 16$. En esta ocasión se hicieron 3 experimentos, cuyos resultados se mostrarán a continuación:

- Experimento 1: variar la cantidad máxima de iteraciones que el algoritmo ejecuta o $maxIterations$. En cada instancia se probaron 4 semillas para cada valor probado de $maxIterations$. Los resultados son los siguientes:

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

- Experimento 2: variar la temperatura inicial del sistema T_0 y el ratio de enfiramiento $coolRatio < 1$ (ratio en que la temperatura disminuye en cada iteración según la fórmula $T = T * coolRatio$).
  En cada instancia se probaron 4 semillas para cada combinación probada de estos parámetros. Los resultados son los siguientes:
  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

- Experimento 3: Variar el valor de $maxR$. En cada instancia se probaron 4 semillas para cada valor probado de $maxR$. Los resultados son los siguientes:
  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

4. Técnica Propuesta - Algoritmo Evolutivo

A continuación, se propone un algoritmo evolutivo para resolver TTP.

Los algoritmos evolutivos [7], o "EA", son métodos inspirados en los procesos evolutivos y de selección natural presentes en la naturaleza, que consisten en trabajar, filtrar, mezclar y modificar distintas "poblaciones" de soluciones en búsqueda de una potencial mejor solución.

En la sección 2.C.4 de este informe, se expuso el algoritmo genético propuesto por [21], el cual usa el framework hadoop para facilitar el proceso. En aquella sección, también se menciona un breve resumen de como funcionan los algoritmos evolutivos.

Básicamente, en cada iteración de un algoritmo evolutivo ocurren los siguientes pasos, que serán explicados más adelante en esta sección:

1) Inicialización de población
2) Selección
3) Cruzamiento
4) Mutación

A. Representación

Hay que dejar en claro que este es un algoritmo evolutivo, más no uno genético. En los algoritmos genéticos [10] existe un proceso de codificación, en el que cada solución se representa como un arreglo "codificado" de datos (números, binarios, letras, etc) llamado "cromosoma"[9], que se puede mezclar y modificar generando nuevas soluciones.

Por ejemplo, el algoritmo de [21] ya mencionado corresponde a un algoritmo genético, pues cada una de sus soluciones corresponde a un arreglo de números $2(N-1)$, donde cada número representa a un posible partido. Por ejemplo, si se tienen 4 equipos: A, B, C y D, la asignación de números por partido se representa en la siguiente matriz, donde cada fila corresponde a los equipos locales de cada partido, y cada fila a los equipos globales:

	A	B	C	D
A		0	1	2
B	3		4	5
C	6	7		8
D	9	10	11	

Fig. 9: Asignación de números por partido usada por [21]

.

Por lo tanto, en esta representación, una posible solución sería el arreglo $[3, 11, 6, 5, 0, 8, 9, 4, 1, 10, 2, 7]$, donde en la primera ronda se jugarían ($\frac{N}{2} = 2$ partidos por ronda) los partidos de $B(L)$ vs $A(V)$ y $D(L)$ vs $C(V)$, en la segunda ronda jugarían $C(L)$ vs $D(V)$ y B vs D, etc.

El problema de usar este tipo de representación para soluciones de TTP es la cantidad de restricciones que este problema presenta en cuanto al orden de sus juegos. Si consideramos que, por ejemplo, cada equipo tiene que jugar en un y solo un partido por ronda, es fácil darse cuenta que no cualquier orden de números es válido. Lo que [21] propone para resolver esta problemática es que cada vez que se genere una solución representada de esta forma mediante cruzamiento y mutación (procesos de EA que serán explicados más adelante), y que no satisfaga las restricciones inherentes de TTP, se descarte ésta.

La implementación de [21] solo se probó para instancias de $N \leq 8$, dando resultados favorables. Sin embargo, si se hubiese probado con instancias de mayor N, el porcentaje de soluciones posibles descartadas por violar restricciones hubiese aumentado debido a la naturaleza combinatoria del espacio de soluciones factibles e infactibles, posiblemente disminuyendo la eficiencia del algoritmo.

Nitin. S. Choubney [5] también implementó un algoritmo genético, usando un método de codificación bastante similar al de [21]. Sin embargo, también fue sólo probado en instancias de $N \leq 8$.

David Moidl[17] por su parte implementó una técnica de codificación distinta para su propuesta de algoritmo genético, que tiene componentes de optimización local. Esta consistía en un arreglo de $2(n-1)$ bloques (cada uno representando una ronda), en los que cada bloque se compone de una permutación de los N equipos.

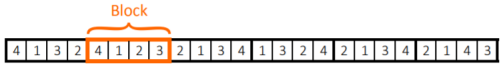


Fig. 10: Ejemplo de cromosoma en el método de [17]

.

La idea principal de la decodificación en [17] (pasar de cromosoma a representación de equipos) es que en cada bloque (es decir, en cada ronda) insertar los equipos en el orden de la permutación del bloque. Para escoger sus rivales, cada equipo tiene un "dominio" de posibles juegos contra cada el resto de equipos otro equipo. Por ejemplo, si $N = 4$, el dominio inicial del equipo 1 es: $\{1(L)v2(V),\ 1(V)v2(L),\ 1(L)v3(V),\ 1(V)v3(L),\ 1(L)v4(V),\ 1(V)v4(L)\}$. Cada vez que un partido se inserte en una ronda, este partido se saca de los dominios de los equipos que juegan en el partido.

Para cada equipo que se pretende insertar en la ronda actual, se inserta el partido de su dominio que minimiza una cierta función costo (de ahí sale la parte de optimización local) y que en el futuro no genere conflictos en la construcción de la solución. Esto último se verifica revisando los dominios de todos los equipos (modificados de acuerdo a la inserción) de distintas formas que no serán detalladas en este informe.

Un posible problema que aparece al usar una representación así es el costo que representa el proceso de codificación, pues ir revisando los dominios de todos los equipos en cada posible inserción de partido para garantizar la construcción de torneos DRR factibles (mas no necesariamente soluciones TTP factibles, por las restricciones atmost y norepeat) demora bastante.

Los algoritmos genéticos son una rama de los algoritmos evolutivos, pero un algoritmo evolutivo que no enfatiza en la codificación de sus soluciones (como el que se está proponiendo) no es genético. Lo que se busca en en este algoritmo propuesto es una representación no codificada que no represente costos de decodificación ni problemas de factibilidad en restricciones duras.

Dicho esto, 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 y poblaciones

Un EA explora conjuntos grandes de soluciones llamados "poblaciones". En cada iteración o "generación", se explora una población distinta. Al final de cada generación, una porción de las soluciones de la población actual se incluirá en la población siguiente. El tamaño de cada población será igual a un parámetro P.

Dicho esto, el resto de soluciones presentes en cada población (en el caso de la población de la primera generación, serían todas las soluciones) serán generadas aleatoriamente mediante el método circular de kirkman[13] para la creación de torneos DRR. Este método también se usó para generar soluciones iniciales en las dos implementaciones propuestas anteriormente por el autor de este informe ([4], [3]). En aquellas ocasiones, el método circular fue explicado de la misma forma en la que se explicará a continuación:

El método circluar 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 emp_0, como en la siguiente figura: (suponiendo un caso $N = 6$)

Fig. 11: Ejemplo de emp_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(emp$_0$,p)*, la cual "fija" uno de los equipos de emp_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:

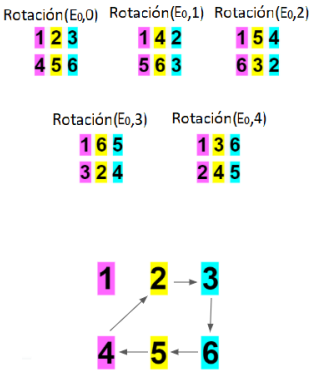


Fig. 12: Ejemplo de rotaciones posibles de emp_0, según distintos valores de p. La forma en que se rotan los equipos de describe abajo. Nótese que: rotación$(emp_0, 0) = emp_0$

.

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(emp_0, p)":

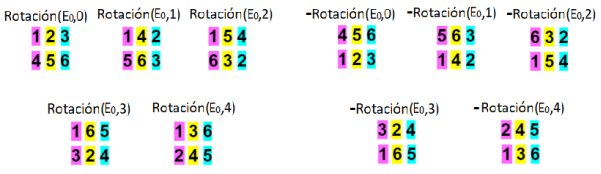


Fig. 13: Ejemplo de rotaciones posibles de emp_0 (a la izquierda), y sus respectivos inversos (a la derecha), en los cuales se invierten los roles de local (tríos de "arriba") y visita (tríos de "abajo").

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
  $emp_0 \leftarrow$ emparejamiento aleatorio inicial de los N equipos.
  $emp \leftarrow \emptyset$
  $DRR \leftarrow$ torneo inicialmente vacío.
  for p in $[0, N-2]$ **do**
    $emp \leftarrow emp \cup$ rotación(emp_0, p)
    $emp \leftarrow emp \cup$ -rotación(emp_0, p)
  end for
  ordenar aleatoriamente emp
  for r in $range(2(N-1))$ **do**
    insertar juegos de $emp[r]$ 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 [18], se explorará en el espacio de soluciones infactibles, al relajar las restricciones atmost y no repeat. Finalmente, hay que aclarar que la solución final que entregue el algoritmo debe ser totalmente factible.

C. Selección y Elitismo

En cada generación, dado el parámetro $0 < E \geq < 1$, se eligen $E * P$ soluciones de la población actual para que sean modificadas en los pasos siguientes del algoritmo. En teoría, las mejores soluciones de la población deberían tener una mayor probabilidad de ser escogidas, para así lograr modificaciones potencialmente mejores. Este proceso es denominado "elitismo"[15].

Por otra parte, las soluciones de la población que no destacan también deberían tener también una probabilidad de ser escogidas, con la esperanza de que desde estas se pueda llegar a soluciones mejores (diversificación).

El método de selección de soluciones será en base a un ranking. Se ordenarán las soluciones de mejor a peor, obteniendo cada solución un ranking según el orden (la mejor solución de la población tiene el ranking "1", la segunda el "2", etc). Para este algoritmo, el criterio de selección se basa en una función costo. Esta función $C(S)$ corresponde a la misma función costo usada en la implementación de simmulated annealing de [1]

(sección 2.C.3) para evaluar y comparar soluciones:

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

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 $w > 0$ es un parámetro peso, cuya función corresponde a indicar cuanto se penalizan las violaciones a las restricciones atmost y norepeat, en el contexto de una búsqueda en el espacio de soluciones infactibles.

De este modo, la mejor solución S_{best} de la población que recibirá el mejor ranking (ranking 1) será la que minimice la función costo. La segunda mejor solución (ranking 2) será la segunda solución con menor costo $C(S)$, etc.

Ahora la pregunta es: "¿Cómo distribuir las probabilidades de cada solución según su ranking?". Una alternativa conocida es la de asignarle a cada solución la siguiente probabilidad:

$$prob(S_i) = \frac{\frac{1}{ranking(S_i)}}{\sum_{j \leq P} \frac{1}{ranking(S_j)}} \quad (2)$$

Un inconveniente de esta alternativa es su costo de implementación, pues si se quiere escoger una solución en base a este criterio, se debe generar un número aleatorio del 0 al 1 e ir buscando la solución escogida acorde a las probabilidades, según el siguiente pseudocódigo:

```
procedure RANDOM_DISTRIBUTION_SELECTION
    x ← número aleatorio entre 0 y 1
    current ← 0
    next ← 0
    for S in poblacion do
        next ← next + prob(S)
        if current ≤ x ≤ next then
            return S_i
        else current ← next
        end if
    end for
end procedure
```

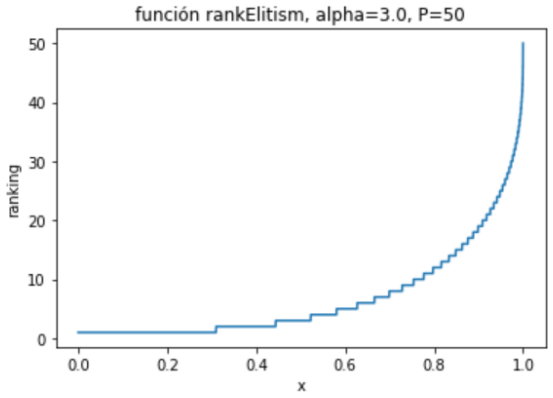
Este procedimiento es de orden O(n) con respecto al tiempo. A continuación se propone otro método de selección de orden O(1), que será implementado en este algoritmo.

Este método consiste en usar la siguiente función, que recibe un número aleatorio x entre 0 y 1 y un parámetro $\alpha \geq 1$:

$$rankElitism(x, P, \alpha) = 1 + redondear(P - (P^\alpha - (1-x)^\alpha))^{\frac{1}{\alpha}} \quad (3)$$

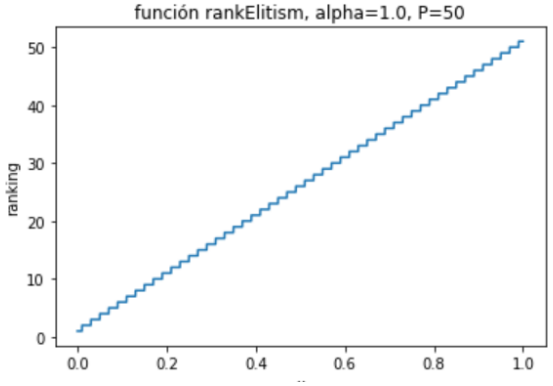
Esta función devuelve el ranking de la solución escogida. Note que entre mayor sea el parámetro α, mayor será el nivel de elitismo, o sea, las probabilidad de que las mejores soluciones sean escogidas.

Por ejemplo, si $\alpha = 3$, y $P = 50$ el gráfico de esta función en su dominio es el siguiente:



Observando la curvatura de la curva, se puede intuír que los rankings mas áltos (1, 2, 3, ...) tienen más probabilidad de ser escogidos por esta función que los rankings más bajos (...,47, 48, 49).

En cambio, si $\alpha = 1$ t $P = 50$, el gráfico se vería así:



Donde la linea es recta y cada ranking tendría la misma probabilidad de ser escogido.

Si se indexan a las soluciones de cada población según su ranking, el proceso de selección será de un tiempo de orden O(1) por cada selección. El pesudocódigo del proceso de selección es el siguiente:

```
procedure SELECCION(poblacion)
    Rankear poblacion de mejor solución a peor
    Psize ← P
    chosens ← []
    for e in range(E * P) do
        x ← random(0, 1)
        rankingS ← rankElitism(x, Psize, α)
        S_ranked ← población con ranking rankingS
        agregar S_ranked a chosens
        quitar S_ranked de poblaciom
        Psize ← Psize − 1
    end for
end procedure
```

D. Cruzamiento

Una vez obtenido el conjunto de $E * P$ soluciones o "conjunto escogido", se aplican los operadores de cruzamiento. Este proceso consiste en elegir varios pares aleatorios de

soluciones “padre” y, desde cada par, aplicar un proceso que genere un nuevo par de soluciones “hijo”.

Tradicionalmente, en el caso de los algoritmos genéticos, el cruzamiento entre dos soluciones[9] se obtiene intercambiando segmentos de sus arreglos de datos codificados (cromosomas) de una forma similar a la que se ilustra en la siguiente imagen:

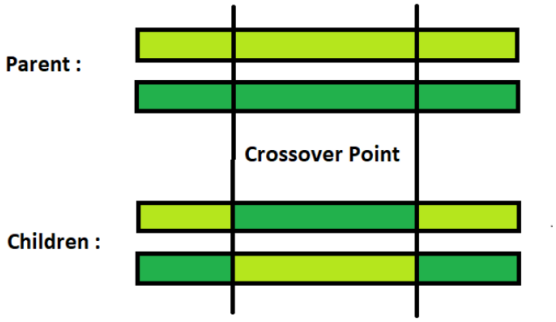


Fig. 14: Ilustración de cruzamiento típico de cromosomas en algoritmos genéticos

Sin embargo en esta implementación, dado que no es un algoritmo genético, se tendrá que ocupar otro método de cruzamiento. Por lo tanto, se propone el siguiente:

Definimos al “mejor equipo de un torneo” como el equipo del torneo que minimiza la función de costo:

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

Donde $d(T)$ es la distancia total de viajes del equipo T en el torneo, $v(T)$ la cantidad de veces que viola las restricciones atmost (jugando más de 3 rondas seguidas de local o más de 3 rondas consecutivas de visita) y norepeat (jugando dos rondas consecutivas contra el mismo equipo) y w es el parámetro “peso”. Esta función costo $C(T)$ es una variación de la función costo usada para evaluar soluciones en el proceso de selección de este algoritmo. La diferencia principal es que lugar de evaluar todo un torneo, evalúa a cada equipo por separado.

Para este operador de cruzamiento se usará la función SwapRounds [1], una de las 5 clases de permutaciones explicadas en la sección 2.C.2, que consiste en intercambiar entre dos rondas, los partidos jugados cada una de las rondas.

El cruzamiento entre dos soluciones o dos “torneos padre” $S_{p,A}$ y $S_{p,B}$ consiste en generar dos nuevos “torneos hijo” $S_{h,A}$ y $S_{h,B}$. Si $T_{best,A}$ y $T_{best,B}$ son los mejores equipos de $S_{p,A}$ y $S_{p,B}$:

- $S_{h,A}$ es una réplica de $S_{p,A}$, a la que se le aplicó $swapRounds$ de tal forma que el equipo $T_{best,B}$ juegue los partidos en el mismo orden que en $S_{p,B}$
- $S_{h,B}$ es una réplica de $S_{p,B}$, a la que se le aplicó $swapRounds$ de tal forma que el equipo $T_{best,A}$ juegue los partidos en el mismo orden que en $S_{p,A}$

De esta forma, se busca que cada torneo adapte el orden de equipos del mejor jugador del otro torneo, tal como en el siguiente ejemplo, en donde $N = 4$:

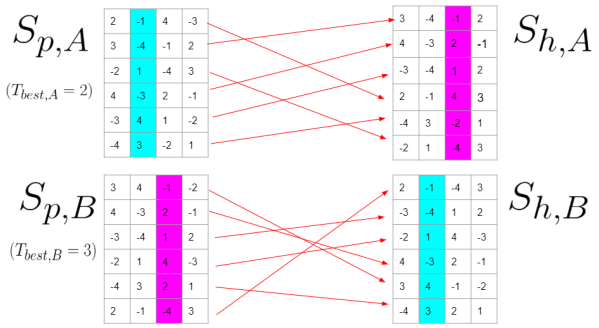


Fig. 15: Ejemplo de cruzamiento de dos torneos

La motivación detrás de este operador de cruzamiento es que las parejas de soluciones intercambien “lo mejor de sí mismas” (los mejores órdenes de juegos para un equipo), para intentar que se generen soluciones “hijas” potencialmente mejores que sus padres.

Luego de generar las dos soluciones con los torneos hijos, estas estas se unen al “conjunto escogido”. Por último, hay que mencionar que las parejas de equipos cruzados se fijan aleatoriamente, donde cada solución del conjunto escogido tiene una probabilidad p_{cross} de pertenecer a alguna pareja de soluciones cruzadas, siendo p_{cross} un parámetro del algoritmo.

El pseudocódigo del operador del proceso de cruzamiento sería el siguiente:

```
procedure CRUZAMIENTO(Conjunto_E)
    revolver Conjunto_E de manera aleatoria
    parereja ← 0

    for i in range(|E * P|) do
        x ← random(0, 1)
        if x < p_cross then
            pareja ← pareja + 1
            if pareja==1 then
                S_{p,A} ← Conjunto_E[i]
                j ← i
            else
                pareja ← 0
                S_{p,B} ← Conjunto_E[i]
                S_{h,A}, S_{h,B} = cruzar(S_{p,A}, S_{p,B})
                Conjunto_E.push(S_{h,A})
                Conjunto_E.push(S_{h,B})
            end if
        end if
    end for
end procedure
```

E. Mutación

Una vez obtenido terminado el proceso de cruzamiento, se tiene “conjunto escogido” con soluciones cruzadas. El último paso de la iteración es la etapa de mutación. En esta etapa, cada solución del conjunto tiene una probabilidad p_{mut} de ser transformada por un “operador de mutación”, siendo p_{mut} un parámetro. A diferencia del cruzamiento, donde se generaban

dos soluciones "hijas" a partir de dos soluciones "padre", la mutación genera una sola solución hija a partir de solo una solución padre (que es reemplazada en el conjunto por la solución hija).

El operador de mutación consiste en formar aleatoriamente parejas de los equipos que participan en el torneo de una solución, e intercambiar entre parejas el orden de partidos usando la función "swapTeams" en un cierto orden aleatorio (primero se aplica swapTeams a una pareja de equipos, luego a otra, etc). SwapTeams es otra de las 5 permutaciones explicadas en las sección 2.C.B, propuestas por [1] para resolver TTP usando Simmulated Annealing.

Un ejemplo de mutación se da en el siguiente torneo:

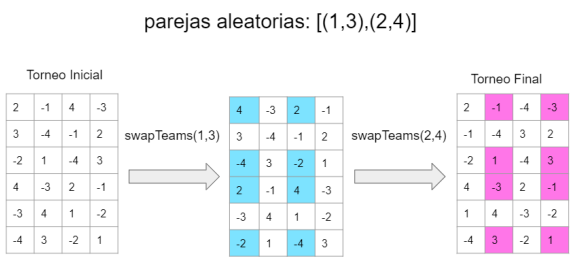


Fig. 16: Ejemplo de Mutación de torneos. Note que el orden de parejas de equipos a la que se le aplica swapTeams es determinante para el resultado final, puesto que en cada "swapTeam", cualquier número de la matriz puede cambiar, independiente de si pertenece a las columnas que se cambian

El pseudocódigo del proceso de mutación es el siguiente:

```
procedure MUTACION(Conjunto_ES)
    for i in range(|Conjunto_ES|) do
        x ← random(0, 1)
        if x < p_mut then
            mutS ← mutar(Conjunto_ES[i])
            Conjunto_ES[i] ← mutS
        end if
    end for
end procedure
```

F. Nuevas generaciones, y proceso general del algoritmo

Una vez terminada la mutación, se procede a la siguiente generación. Todo el "conjunto escogido" cuyas $E * P$ soluciones pudieron haber sido cruzadas y/o mutadas. Aquellas soluciones serán incluidas en la siguiente población, de la siguiente generación.

Además de estas $E * P$ soluciones, también se incluirán en la nueva población $(E - 1) * P$ nuevas soluciones generadas aleatoriamente por el método circular ya explicado.

Cabe destacar que cada vez que se genera una solución con el método circular, siendo producto de cruzamiento o como resultado de una mutación se revisa si es la "mejor solución factible" encontrada hasta el momento. La mejor solución factible es la solución cuyo torneo tiene 0 violaciones a las restricciones atmost y norepeat (por eso es factible) y tiene la menor distancia total recorrida por los equipos.

Una vez se completan G generaciones (siendo G un parámetro), el algoritmo retorna a la mejor solución encontrada en toda la ejecución.

El pseudocódigo general de todo el proceso es el siguiente:

```
procedure EVOLUTIONARY
    poblacion ← []
    for g in range(|G|) do
        while |poblacion| < P do
            newS ← INITIAL_DRR()
            agregar newS a poblacion
        end while
        Conj_E ← SELECCION(poblacion)
        Conj_E ← CRUZAMIENTO(Conj_E)
        Conj_E ← MUTACION(Conj_E)
        poblacion ← Conj_E
    end for
    retornar mejor solución factible encontrada
end procedure
```

G. Parámetros

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

- $\alpha \geq 1$: parámetro usado por función $RANDOM\_DISTRIBUTION\_SELECTION$. Representa el nivel de elitismo en el proceso de selección.
- $G > 0$: número de generaciones que el algoritmo ejecutará
- $P > 0$: tamaño de población en cada generación
- $0 < E \leq 1$: Porción de los individuos de la población escogidos en cada generación para participar en los procesos de cruzamiento y población.
- $0 \geq p_{cross} \geq 1$: probabilidad de cruzamiento.
- $0 \geq p_{mut} \geq 1$: probabilidad de mutación.
- $w \geq 0$: variable peso, usado en las funciones costo C presentes en el proceso de selección y de cruzamiento.

5. Escenario experimental

Se probará una implementación del algoritmo propuesto 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.

Los experimentos que se realizarán consistirán en variar los parámetros del algoritmo, el cual inicialmente recibirá los siguientes "parámetros base":

- $\alpha = 2$. Debido a que el método de selección que usa este parámetro es nuevo, no hay estudios que recomienden un valor base para este parámetro. Se escogió este valor pues se estimá que logrará un buen equilibrio entre diversificación y elitismo.
- $G = 2000$. En algunas implementaciones de algoritmos evolutivos para TTP ([21], [17]) se usan cantidades de

generaciones del orden de los miles. Para esta implementación, se escogieron 2000 generaciones por ejecución pues es se estima que es un valor relativamente alto (en el sentido de que sería suficiente para que el algoritmo converja a una solución de calidad aceptable), pero que no compromete demasiado tiempo de ejecución.
- P = 1000. Similar a lo que ocurre con G, hay implementaciones [21] que usan tamaños de población del ordenes de mil. En este caso, se espera que 1000 soluciones por población sea suficiente para que en cada generación se trabaje con una población diversa.
- E = 0.1. Se escogió este valor a partir de la implementación de [17], la cual escoge 10 individuos en cada población de 100 soluciones (un 10%).
- p_{cross} = 1. Se escogió este valor a partir de la implementación de [19], en la cual todas las soluciones escogidas en cada generación se cruzan.
- p_{mut} = 0.3. Se escogió este valor a partir de la implementación de [17], en la cual las soluciones tienen un 30% de probabilidad de mutar.
- w = 9000. Se escogió este valor debido a que en la implementación de [1], de la cual se sacó la función costo C que usa w, uno de los experimentos consistía en usar un valor inicial de w de 18000. Sin embargo, en esa implementación el valor de w disminuía con el número de iteraciones en cada ejecución, por lo que se estima un valor promedio de $w = \frac{18000}{2} = 9000$ durante la ejecución.

Se realizarán tres experimentos. En cada uno de estos experimentos se variarán algunos de los parámetros base por separado

A. Experimento 1 - Cantidad de individuos

Para este experimento, se ejecutará el algoritmo varias veces, variando los valores de los parámetros P y E de la siguiente forma:

- P = 1000, E = 0.1 (base)
- P = 2000, E = 0.1
- P = 1000, E = 0.2
- P = 2000, E = 0.2
- P = 1000, E = 0.3
- P = 2000, E = 0.3

Estos dos parámetros se modificarán en el mismo experimento, pues los dos están relacionados a la cantidad de individuos que se tienen y que se eligen desde cada población.

Los valores del resto de parámetros corresponderán a los valores base.

B. Experimento 2 - probabilidades de transformación

Para este experimento, se ejecutará el algoritmo varias veces, variando los valores de los parámetros p_{cross} y p_{mut} de la siguiente forma:

- p_{cross} = 1, p_{mut} = 0.3 (base)
- p_{cross} = 1, p_{mut} = 0.6
- p_{cross} = 0.5, p_{mut} = 0.3
- p_{cross} = 0.5, p_{mut} = 0.6

Estos dos parámetros se modificarán en el mismo experimento, pues los dos están relacionados al mismo proceso: la transformación de soluciones. Los valores del resto de parámetros corresponderán a los valores base.

C. Experimento 3 - nivel de elitismo

Para este experimento, se ejecutará el algoritmo varias veces, variando el valor del parámetro α de la siguiente forma:

- α = 2.0
- α = 3.0

Los valores del resto de parámetros corresponderán a los valores base.

D. Mediciones e instancias

Para cada uno de los tres experimentos se medirá el desempeño del algoritmo de dos formas distintas:

La primera forma medición consistirá en medir el comportamiento del algoritmo en relación a las mejores soluciones encontradas durante cada ejecución según la generación, además del tiempo de ejecución.

Para esto, se probará en las instancias $NL6$, $NL10$, $NL14$, variando los parámetros según el experimento. Se escogieron estas tres instancias pues sus valores de N difieren mucho entre sí, y se busca estudiar el comportamiento y desempeño del algoritmo según N.

Para cada experimento, en cada configuración de parámetros y en cada instancia se ejecutará el algoritmo una sola vez (una semilla). Para cada ejecución, se mostrará su gráfico de convergencia (mejores soluciones encontradas durante cada generación), además del tiempo total de ejecución y la distancia total (función objetivo) de la mejor solución encontrada.

La segunda forma de medición buscará medir el comportamiento del algoritmo cuando este se ejecuta varias veces para una misma configuración de parámetros.

Para esto, en cada configuración de parámetros de cada experimento se probará el algoritmo 10 veces (10 semillas) en la instancia $NL10$ (solo se uso una instancia pues el tiempo de experimentación hubiese sido mucho más alto en el caso contrario). Para cada configuración de parámetros, se mostrarán gráficos de dispersión en forma de boxplots con respecto a las distancias totales de la mejor solución obtenida en cada ejecución, a parte del promedio de esas distancias totales. En cada experimento, se compararán los resultados de cada configuración.

En esta segunda forma de medición, no se medirán los tiempos de ejecución pues estos ya fueron obtenidos para la primera medición (la de la única semilla por configuración). No hay indicios en el algoritmo para plantear la hipótesis

de que el tiempo de ejecución puede variar significativamente cuando los valores de los parámetros se mantienen constantes. Incluso cuando algunas de las decisiones que toma el algoritmo son aleatorias, las probabilidades de tomar esas decisiones se basan en parámetros del algoritmo (p_{cross} y p_{min}). Por la ley de los grandes números, se espera a que si estas probabilidades se mantienen entre varias ejecuciones, la cantidad de decisiones aleatorias tomadas no variará lo suficiente para alterar los tiempos de ejecución de manera significativa.

Es necesario mencionar que el objetivo de todo este escenario experimental **no** es encontrar una configuración óptima de parámetros para este algoritmo (es decir, una configuración de parámetros que lleve a las mejores soluciones obtenibles a partir de esta implementación), sino que se busca estudiar el comportamiento y el desempeño del algoritmo en relación a la variación de los parámetros.

Por último, el valor base de w (9000) no se cambiará en todos los experimentos, pues no es un parámetro tan relacionado a los algoritmos evolutivos en sí, sino que se relaciona a la evaluación de las soluciones según el "nivel de factibilidad" (relajada para las restricciones atmost y norepeat) de cada solución.

6. Resultados obtenidos

A. *Experimento 1*

Ejecución de una semilla:

Para NL6:

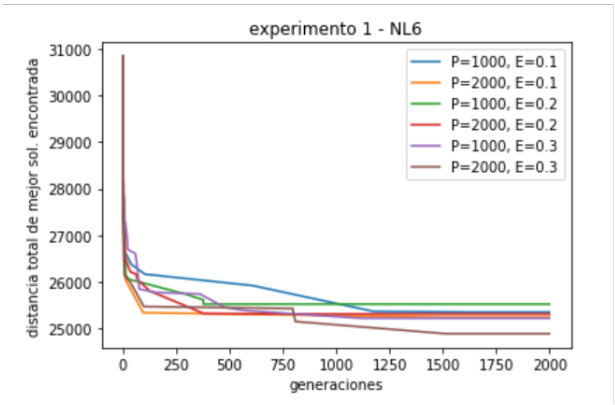


Gráfico de convergencia exp1-6: Convergencia de mejores soluciones encontradas por generación, en el experimento 1, para la instancia NL6.

Para NL10:

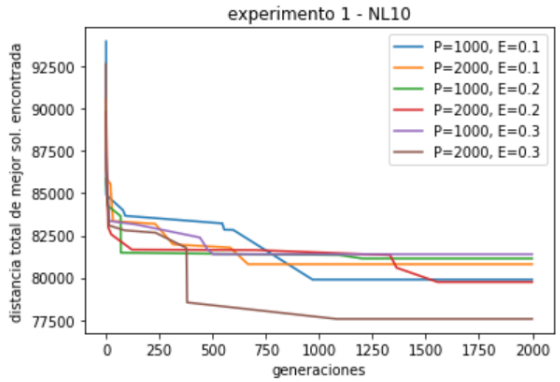


Gráfico de convergencia exp1-10: Convergencia de mejores soluciones encontradas por generación, en el experimento 1, para la instancia NL10.

Para NL14:

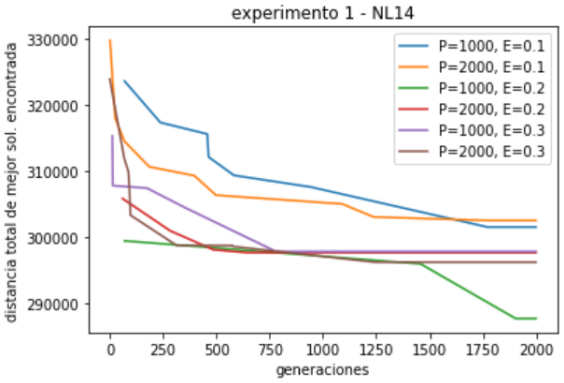


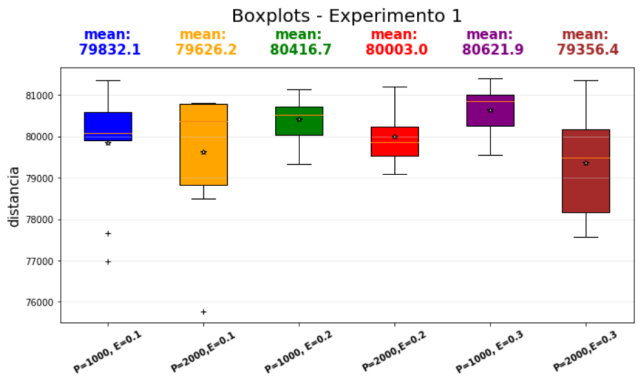
Gráfico de convergencia exp1-14: Convergencia de mejores soluciones encontradas por generación, en el experimento 1, para la instancia NL14.

La siguiente tabla muestra los tiempos de ejecución y distancias totales obtenidas de cada configuración, en cada instancia:

Params.	Medida	NL6	NL10	NL14
P = 1000	Mejor Distancia	25353	79897	301483
E = 0.1	Tiempo [S]	31.11	81.53	160.32
P = 2000	Mejor Distancia	25244	80805	302474
E = 0.1	Tiempo [S]	64.97	170.23	306.39
P = 1000	Mejor Distancia	25523	81141	287661
E = 0.2	Tiempo [S]	32.14	82.68	146.81
P = 2000	Mejor Distancia	25311	79754	297625
E = 0.2	Tiempo [S]	65.89	171.21	304.79
P = 1000	Mejor Distancia	25221	81393	297790
E = 0.3	Tiempo [S]	35.41	83.12	143.62
P =2000$	Mejor Distancia	24889	77573	296180
E = 0.3	Tiempo [S]	71.30	165.84	300.11

tabla exp1

Ejecución de varias semillas



Boxplots exp1: boxplots de resultados de experimento 1, para la instancia NL10. Para cada configuración de parámetros se realizó un muestreo de 10 resultados. Arriba se muestran los promedios de distancias totales obtenidas para cada configuración de parámetros.

Análisis de los resultados del experimento 1

A partir de los gráficos de convergencia exp1-10 y exp1-14, se puede inferir que, cuando el valor de N es lo suficientemente grande ($N > 6$), Un incremento en P ($P = 2000$) cuando E es constante podría llevar a mejores soluciones en generaciones más tempranas. Sin embargo, un cuando P es menor ($P = 1000$), el algoritmo encuentra mejores soluciones en las últimas generaciones que cuando P es grande. Esta inferencia proviene que, cuando $N > 6$ entre curvas de convergencia de igual E (azul y amarilla, verde y roja; y purpura y café), las curvas de $P = 2000$ suelen mantenerse más abajo que las curvas de $P = 1000$, pero estas últimas siempre alcanzan mejores soluciones en las últimas generaciones.

Por otro lado, no parece existir una indicación de que la variación del parámetro E implique un comportamiento distintivo, pues en los tres gráficos de convergencia la diferencia de comportamiento entre curvas de igual P pero distinto E parece ser distinta.

Analizando la tabla exp1, se observa que los tiempos de ejecución son más grandes cuando el valor de N es más grande. Esto no sorprende, pues los tiempos que demoran los operadores de cruzamiento, mutación, evaluación y generación de soluciones claramente dependen de N.

Además, las diferencias de tiempos de ejecución varían mucho entre ejecuciones de igual N y E, pero distinto P. Los tiempos de ejecuciones cuando $P = 2000$ parecen ser del doble que los tiempos de ejecuciones cuando $P = 1000$. Una posible hipótesis de porque ocurre esto, es porque cuando el valor de E se mantiene igual y el valor P se incrementa al doble, la cantidad de soluciones que hay que seleccionar, mutar y cruzar también incrementa al doble (pues E representa un porcentaje de P, no una cantidad de soluciones). Sin embargo, se puede rechazar esta hipótesis pues en instancias de igual P pero distinto E los tiempos de ejecución no varían significativamente, lo que indica que los procesos de cruzamiento y mutación demoran casi lo mismo cuando se procesan $E * P$ soluciones que cuando se procesan $(2 * E) * P$ soluciones. A partir de este análisis, se infiere que los procesos más costosos del algoritmo en cuanto a tiempo corresponden a la selección y a la generación de $1-E*P$ soluciones iniciales al inicio de cada generación (en contraposición con los procesos de cruzamiento y mutación).

Analizando los boxplots del experimento 1, se puede observar que cuando el valor de P aumenta, la calidades de las soluciones obtenidas no solo son mejores (pues los boxplots se concentran más abajo, ya que las distancias son menores, además de que los promedios son menores), sino que son más diversas, pues los boxplots en sí son más extensos, tanto entre los cuartiles centrales como en los cuartiles ”exteriores”. Además, la cantidad de soluciones outliers parece ser mayor cuando $E = 0.1$ (pues solo se presentan en el boxplot azul y en el boxplot naranjo).

Por otro lado, del hecho de que las calidades de las soluciones obtenidas son más diversas cuando $P = 2000$ se podría inferir de que la hipótesis hecha a partir de los gráficos de convergencia (la cual especulaba que el algoritmo encuentra mejores soluciones cuando P es mayor, pero demora más generaciones en superar al desempeño obtenido cuando P es menor) no necesariamente se cumple en todos los casos.

B. Experimento 2

Ejecución de una semilla

Para NL6:

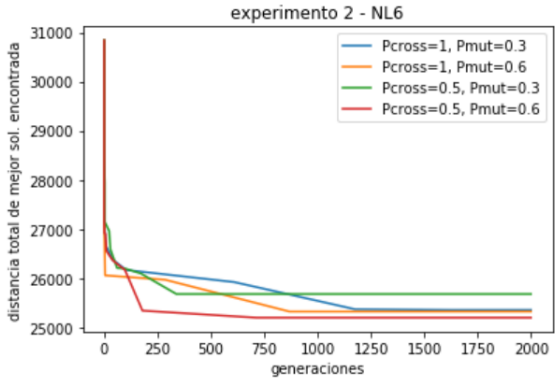


Gráfico de convergencia exp2-6: Convergencia de mejores soluciones encontradas por generación, en el experimento 2, para la instancia NL6.

Para NL10:

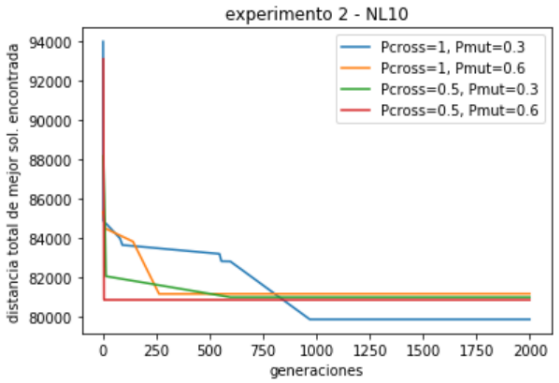


Gráfico de convergencia exp2-10: Convergencia de mejores soluciones encontradas por generación, en el experimento 2, para la instancia NL10.

Para NL14:

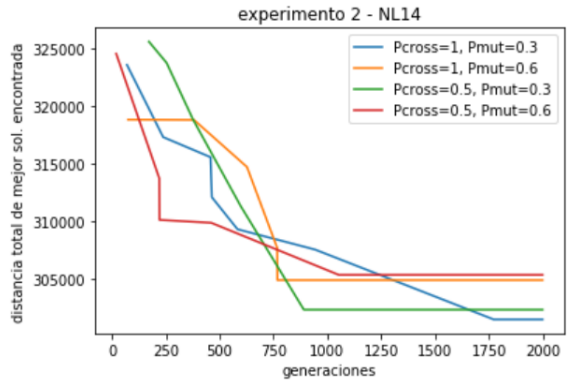


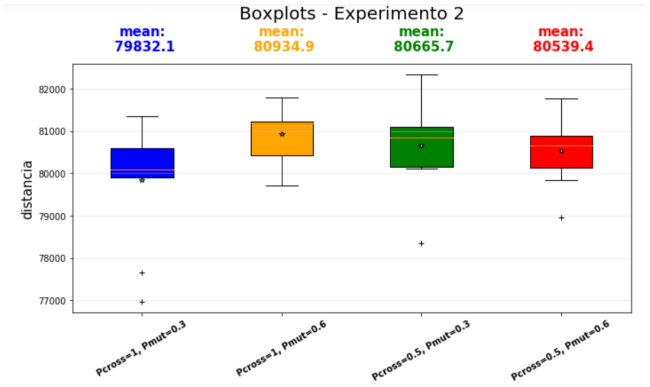
Gráfico de convergencia exp2-14: Convergencia de mejores soluciones encontradas por generación, en el experimento 2, para la instancia NL14.

La siguiente tabla muestra los tiempos de ejecución y distancias totales obtenidas de cada configuración, en cada instancia:

Params.	Medida	NL6	NL10	NL14
p_{cross} = 0.1 p_{mut} = 0.3	Mejor Distancia	25353	81144	301483
	Tiempo [S]	31.11	81.53	160.32
p_{cross} = 1 p_{mut} = 0.6	Mejor Distancia	25321	81196	304891
	Tiempo [S]	32.73	86.31	160.89
p_{cross} = 0.5 p_{mut} = 0.3	Mejor Distancia	25677	81021	302328
	Tiempo [S]	30.61	84.35	148.47
p_{cross} = 0.5 p_{mut} = 0.25	Mejor Distancia	25196	80885	305344
	Tiempo [S]	31.09	83.79	166.32

tabla exp2

Ejecución de varias semillas



Boxplots exp2: boxplots de resultados de experimento 2, para la instancia NL10. Para cada configuración de parámetros se realizó un muestreo de 10 resultados. Arriba se muestran los promedios de distancias totales obtenidas para cada configuración de parámetros.

Análisis de los resultados del experimento 2

En los gráficos de convergencia se observa que el desempeño la curva ruja, la cual representa ejecuciones con menores probabilidades de cruzamiento, pero mayor probabilidad de mutación, va disminuyendo a medida que aumenta N, pues para $N = 6$ es la que se mantiene en mejores soluciones por casi todo el tiempo, en $N = 10$ su comportamiento se asemeja con dos de las otras tres curvas y para $N = 14$ ya es superada por todas las otras ejecuciones en la generación 1250 (aproximadamente).

Por otro lado, cuando $N = 10$ y $N = 14$ la curva con mayor desempeño final es la curva azul, que representa una baja probabilidad de mutación pero una probabilidad máxima de cruzamiento. Además, cuando $N = 14$ la segunda curva que parece mantenerse en soluciones de mayor calidad es la verde, que representa bajas probabilidades tanto de cruzamiento como de mutación.

A partir de este último análisis, se puede inferir que el operador de mutación no es tan "valioso" como el operador de cruzamiento, pues para valores de N grandes ($N > 6$) las ejecuciones de mayor desempeño son las que conllevan una probabilidad baja de mutación. Una hipótesis de porqué ocurre esto se debe al echo de que, a diferencia del operador de cruzamiento, del cual las soluciones hijas se unen a las soluciones padre para la siguiente generación, las soluciones que son mutadas son modificadas permanentemente, independientemente de si las calidades de estas aumentan o disminuyen. Sería útil agregar a la implementación la opción de "retroceder" cada mutación en caso de que la solución mutada sea de menor calidad que como era originalmente.

A partir de la tabla exp2, no existe evidencia suficiente para indicar que los tiempos de ejecución varían significativamente cuando los parámetros p_{cross} y p_{mut} cambian, lo que encaja con la afirmación realizada anteriormente, la cual indicaba que los procesos de mutación y cruzamiento no son muy significativos en cuanto al tiempo de ejecución en comparación con los procesos de selección y generación de $(1 - E) * P$ soluciones iniciales al inicio para cada población.

A partir del boxplot del experimento 2, se observa que

las calidades de soluciones de los distintos boxplots se concentran en regiones muy similares, especialmente los rangos intercuartílicos. Gran parte de estos rangos intercuartílicos se mantienen entre 80000 y 81000. Aparentemente, los mejores resultados se obtienen cuando $p_{cross} = 1$ y $p_{mut} = 0.3$, pues el máximo y el rango intercuartílico del boxplot azul se mantienen en valores más bajos que en los otros boxplots, además de que se presentan dos soluciones outliers de distancia menor a 78000.

La configuración de parámetros que lleva a las segunda mejor distribución de soluciones (en cuanto a distancia) es $p_{cross} = 0.5$, $p_{mut} = 0.6$, pues tanto su mínimo es el mínimo más bajo y su máxmo es el segundo máximo más bajo de todos los boxplots (además, se considera mejor que el boxplot verde pues el máximo de este es el más alto en cuanto a distancia). Esto podría contradecirse con el análisis hecho a partir de los gráficos de convergencia, pues de este se infería que el algoritmo tiene un menor desempeño cuando $p_{cross} = 0.5$ y $p_{mut} = 0.6$. Sin embargo, esa hipótesis se realizó principalmente en base a los resultados obtenidos a partir de la instancia $NL14$ y no $NL10$, instancia de la cual se obtuvieron los boxplots.

C. Experimento 3

Ejecución de una semilla: Para NL6:

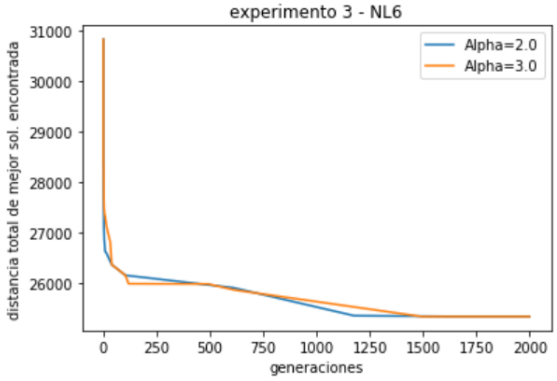


Gráfico de convergencia exp3-6: Convergencia de mejores soluciones encontradas por generación, en el experimento 3, para la instancia NL6.

Para NL10:

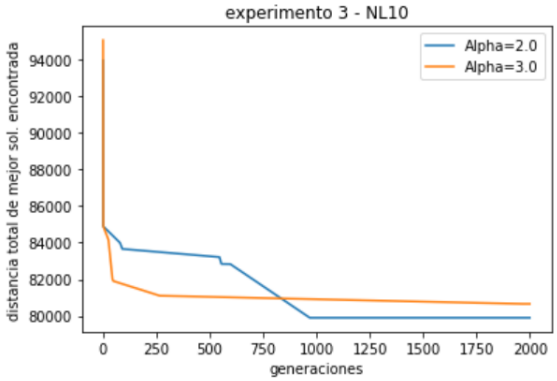


Gráfico de convergencia exp3-10: Convergencia de mejores soluciones encontradas por generación, en el experimento 3, para la instancia NL10.

Para NL14:

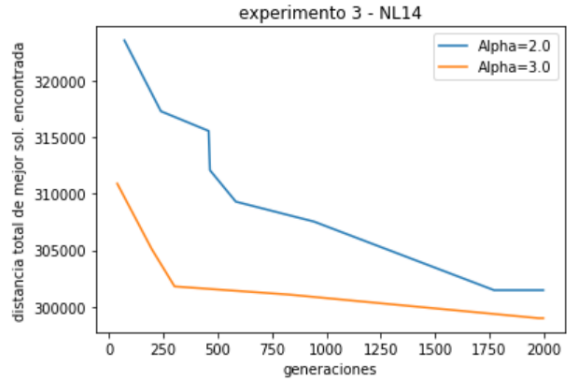
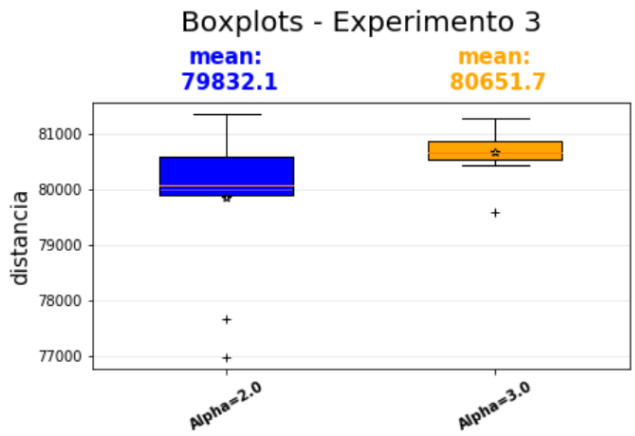


Gráfico de convergencia exp3-14: Convergencia de mejores soluciones encontradas por generación, en el experimento 3, para la instancia NL14.

La siguiente tabla muestra los tiempos de ejecución y distancias totales obtenidas de cada configuración, en cada instancia:

Params.	Medida	NL6	NL10	NL14
$\alpha = 2.0$	Mejor Distancia	25353	81144	301483
	Tiempo [S]	31.11	81.53	160.32
$\alpha = 3.0$	Mejor Distancia	25349	80657	299009
	Tiempo [S]	33.37	82.89	154.57

Ejecución de varias semillas



Boxplots exp3: boxplots de resultados de experimento 3, para la instancia NL10. Para cada configuración de parámetros se realizó un muestreo de 10 resultados. Arriba se muestran los promedios de distancias totales obtenidas para cada configuración de parámetros.

Análisis de los resultados del experimento 3

A partir de los gráficos de convergencia se puede observar que a medida que aumenta N, el comportamiento del desempeño del algoritmo cuando $\alpha = 2.0$ y cuando $\alpha = 3.0$ difieren cada vez más, independiente de cual entregue mejores resultados. En la instancia $NL6$, ambas curvas de convergencia se comportan de manera muy similar. En $NL10$, la ejecución del algoritmo cuando $\alpha = 2.0$ difiere bastante en comportamiento que la ejecución cuando $\alpha = 3.0$, considerando que inicialmente la última encuentra soluciones mucho mejores, pero entre las generaciones 750 y 1000 es superada por la primera. La mayor diferencia de comportamientos ocurre en $NL14$, pues las curvas se mantienen alejadas entre sí en todo momento.

La razón de porqué ocurre esto se puede teorizar analizando los boxplots del experimento 3, pues la dispersión de resultados es mucho menor cuando $\alpha = 3.0$ que cuando $\alpha = 2.0$, pues los el rango intercuartil del boxplot naranja ($\alpha = 2.0$) y la distancia entre su mínimo y su máximo es mucho menor que en el boxplot azul ($\alpha = 3.0$). Esto podría significar que el comportamiento del algoritmo en cuanto a soluciones encontradas es mucho menos variado cuando el nivel de elitismo es mayor, probablemente debido a que en este caso el algoritmo elige transformar y experimentar desde sólo soluciones mejores, dejando de lado a las soluciones de menor calidad que podrían llevar a soluciones de mayor o menor calidad, dependiendo del caso. Quizás es por esto que el comportamiento de ambas curvas difiere tanto en $N = 10$ y $N = 14$, instancias de N grande, donde la diferencia entre elitismo y diversificación es más notoria que en $N = 6$ debido a que el espacio de búsqueda es mucho más grande.

A partir de la tabla, no parece existir evidencia suficiente que el cambio de valor del parámetro α tenga un impacto significativo en los tiempos de ejecución.

D. Análisis general de los resultados y comparación con implementaciones anteriores

Al comparar los resultados de este experimento con los obtenidos a partir de la implementación del algoritmo greedy realizada anteriormente (sección 3.A), es posible darse cuenta de que el desempeño del algoritmo evolutivo propuesto es generalmente mejor en cuanto a distancias totales de funciones obtenidas, considerando que los resultados promedio de las 1000 ejecuciones realizadas por instancia con el algoritmo greedy fueron menores a cualquiera de las distancias obtenidas por el algoritmo evolutivo presentes en las tablas exp1, exp2 y exp3.

Sin embargo, observando las menores distancias obtenidas en cada una de las 1000 ejecuciones por instancia del algoritmo greedy (es decir, las distancias de las mejores soluciones obtenidas), se obtiene que estas son menores a algunas de las distancias de las soluciones obtenidas por el EA presentes en las tablas exp1, epx2 y exp3 cuando $N < 6$. Considerando además que los tiempos de ejecución que demoró la implementación de greedy en generar 1000 soluciones fueron bastante menor a los tiempos de ejecución de la implementación de EA (para greedy demoró menos de 1 minuto para $NL6$ y $NL10$ y menos de dos minutos en $NL14$, mientras que en el EA demoraron mucho más), se puede concluir que el algoritmo evolutivo no necesariamente es una mejor opción para resolver TTP que simplemente generar 1000 soluciones usando el método greedy.

Por otro lado, si se tiene en mente que los resultados de la implementación de simmulated annealing (sección 3.B) fueron notoriamente mejores a las de greedy [3], no sorprende que las soluciones obtenidas por SA sean mucho menores a las de EA, pues todas las distancias totales de soluciones obtenidas por SA fueron notoriamente menores a las obtenidas por EA, presentes en las tablas exp1, exp2 y exp2 (aunque para el caso de $NL6$ no está claro, pues esta instancia no fue probada en la implementación de EA). Además, los tiempos de ejecución fueron mucho menores en SA que en EA, pues incluso para cuando se ejecutó el mayor número de iteraciones en la instancia $NL14$ para el algoritmo SA, el tiempo de ejecución nunca superó al medio minuto, mientras que los tiempos de ejecución de EA fueron siempre mayores a un minuto, cuando $N > 6$ (de nuevo, el caso de $NL6$ no está claro, pero se podría inferir el tiempo de ejecución de SA sería mucho menor si se probara $NL6$ que EA, considerando que los tiempos de ejecución de EA para $NL6$ fueron mayores a los de SA incluso para $NL14$).

7. Conclusiones

Los algoritmos evolutivos proponen una exploración menos ”guiada”, pero mucho más diversa, pues en lugar de ir mejorando una sola solución, se elige explorar desde el procesamiento de una cantidad grande de soluciones.

Para que un algoritmo evolutivo sea efectivo es necesario que la naturaleza de la representación de sus soluciones facilite operar sobre éstas de tal modo que se garantice un alto nivel de

diversificación al momento de cruzarlas o mutarlas a medida que se itera de generación a generación.

En este informe, se propuso y se implementó un algoritmo evolutivo el cual, lejos de enfocarse en representar sus soluciones de una manera codificada y compacta, se centró en usar una representación más "familiar" de estas, con el fin de no gastar recursos (tiempo de ejecución) en los procesos de codificación y decodificación de soluciones.

Sin embargo, los resultados no fueron bastante buenos, en comparación con otras implementaciones realizadas anteriormente (ver secciones 2 y 3) de metaheurísticas diferentes. Una hipótesis del bajo desempeño demostrado por esta implementación se encuentra justamente en la representación no codificada de las soluciones.

Es posible que cruzar y mutar matrices de la forma propuesta en este informe no implique un nivel adecuado de diversificación o una exploración más efectiva para llevar a mejores soluciones. [17] implementó un EA usando una representación más codificada de soluciones (explicada brevemente en la sección 4.A),

	TTPMA			
Instance	Min	Avg	Std. Dev	Avg. Time
NL10	65 560	66 069.0	288.769	1 466.3
NL12	127 266	128 842.3	766.074	3 060.4
NL14	227 921	230 984.1	1 528.669	7 361.0
NL16	322 876	326 704.8	2 458.088	11 740.9

Según esta tabla, [17] obtuvo resultados mucho mejores en cuanto a distancias totales. Por otro lado, sus tiempos de ejecución fueron mucho mayores (más de 1000 segundos para $NL10$), principalmente debido al alto tiempo demandado por el método de decodificación que propuso, entre otros componentes propios de su implementación.

Quizás sea imposible implementar un EA para TTP (genética o no) que logre un buen equilibrio entre buenas funciones objetivo obtenidas y cortos periodos de ejecución, en comparación con otras metaheurísticas (tanto en funciones objetivo como en tiempo). Según quienes propusieron TTP [11], este es un problema cuya complejidad aumenta de una manera extremadamente al pasar a instancias de N (no mucho) mayor, por lo que es posible que sea mejor recurrir a métodos más "guiados" u otras metaheurísticas que EA para resolverlo o llegar a soluciones aceptables. Esto último no quiere decir que los algoritmos evolutivos no sean la mejor opción en casos generales, pero para este problema en particular quizás sea mejor optar por otros métodos que EA, como los mencionados en este informe.

Otras hipótesis con respecto al bajo desempeño de esta implementación consisten en un operador de mutación no tan efectivo, o al hecho de que no se experimentó ni varió el parámetro peso w, pues el proceso de selección podría ser más efectivo si se regula este valor según el tamaño de la instancia mientras se ejecuta el algortimo.

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. Simmulated annealing en travelling tournament problem. 2020.

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

[5] Nitin S. Choubey. A novel encoding scheme for traveling tournament problem using genetic algorithm. *IJCA*, (2):79–82, 1984.

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

[7] David B. Fogel and Lawrence J. Fogel. An introduction to evolutionary programming. *Artificial Evolution*, pages 21–23, 1995.

[8] 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".

[9] Oğuzhan Hasançebi and Fuat Erbatu. Evaluation of crossover techniques in genetic algorithm based optimum structural design. *Computers & Structures*, 78:435–448.

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

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

[12] 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.

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

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

[15] Eckart Zitzler Marco Laumanns and Lothar Thiele. On the effects of archiving, elitism, and density based selection in evolutionary multi-objective optimization. *Evolutionary Multi-Criterion Optimization*, pages 181–196, 2001.

[16] 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.

[17] David Moidl. Solving scheduling problems using evolutionary algorithm. 2015.

[18] 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.

[19] Tinnaluk Rutjanisarakul and Thiradet Jiarasuksakun. A sport tournament scheduling by genetic algorithm with swapping method. *Journal of Engineering and Applied Sciences*.

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

[21] 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.

[22] 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”.

[23] 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.