# Travelling Tournament Problem: un acercamiento Greedy

Yoel Berant Elorza, Universidad Técnica Federico Santa María

Mayo 2020

**Abstract**

"Greedy Search" (o sólo greedy) es una heurística simple para solucionar problemas combinatorios. Esta consiste en, desde un "punto inicial", generar una solución factible paso a paso según una función miope la cual busca elegir una transformación localmente óptima, todo esto con el fin de intentar minimizar (o maximizar, según el problema al que se enfrente) una función objetivo. Si bien greedy no es la heurística que a mejores resultados suele llegar en muchos problemas, es la más simple de implementar y puede ser un punto de partida para desarrollar métodos más complejos.

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

Cabe destacar que el algoritmo greedy propuesto no busca encontrar la mejor solución factible (la que minimice la distancia total recorrida), sino que pretende ser la piedra angular de lo que en el futuro podría ser una implementación más compleja y eficaz.

## 1 Introducción

El "travelling tournament problem" (o TTP) [6] 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 y, finalmente, se propone y se implementa un método greedy para entregar una solución al problema.

# 2 Estado del arte

## 2.1 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 [4].

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 [13].

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. [6] 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.

## 2.2 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 [6], 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.

## 2.3 Soluciones Planteadas

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

### 2.3.1 Solución Original

Kelly Easton, George Nemhausery Michael A. Trick, las mismas personas que definieron TTP [6] propusieron un método para resolverlo basado en constraint programming, integer programming y branch and bound [7].

La motivación detrás de esta decisión en cuanto a paradigmas [7] 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 2.2). 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.

### 2.3.2 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 [11] 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 objetivo). 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, lo cual no implica 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" [6], 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 |

Figure 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 |

Figure 2: tiempos de ejecución en segundos de las implementaciones de simmulated annealing [1]

### 2.3.3 Simmulated Annealing con coloreos de grafo

En el año 2012, Sevnaz Nourollahi, Kourosh Eshghi y Hooshmand Shokri Razaghi [10] 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 los cuales cada vértice representa un posible juego o partido ($\{i, j\}$, en el que $i$ juega como local e $j$ como visita) y la unión entre dos vértices representa la imposibilidad de que dos partidos 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 $l$)[8].

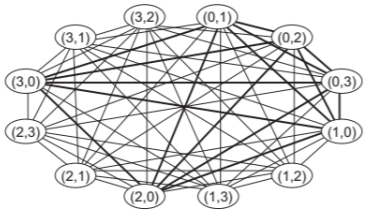


Figure 3: representación de juegos de un DRR a través de un grafo, con N=4

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 que en este caso, sirve como solución de TTP (ignorando las restricciones norepeat y atmost).

Usando esta representación, se propone añadir a la implementación (con respecto 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*[8], 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.

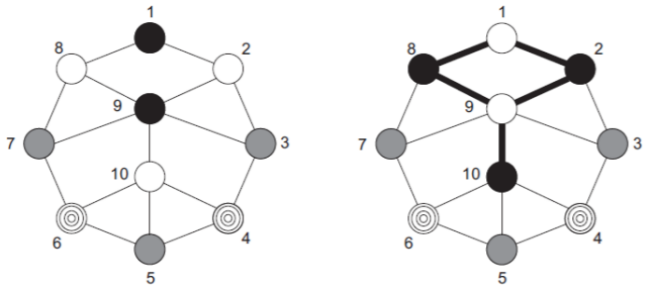


Figure 4: ejemplo de intercambio de colores en cadena kempe[10].

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 [10].

Para probar esta implementación se usaron instancias del benchmark "Major League Baseball" [3]. 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 [10] alcanza mejores soluciones en un tiempo menor a las demás.

### 2.3.4 Algoritmos genéticos con Hadoop

Laxmi Thakare, Dr. Jayant Umale y Bhushan Thakare [12] proponen en el año 2012 una implementación basada en algoritmos genéticos (GA).

Los GA [5] son métodos de búsqueda basados en los procesos evolutivos y de selección natural presentes en los organismos de la vida real. Los GA seleccionan la mejor de muchas soluciones generadas a traves de los siguientes pasos:

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

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

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

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

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

La motivación detrás de esta implementación, según argumentan sus autores [12], 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" [6] 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 2.3.2), 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).

### 2.3.5 Autómata de aprendizaje

Mustafa Misir, Tony Mauters, Katja Veerbeck y Greet Vanden Berghe[9] 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 2.3.2)[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 [9] 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 |

Figure 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 |

Figure 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** | – | – |

Figure 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 2.3.1 y 2.3.2)), LA resulta ser un excelente método para resolver TTP, considerando que según los desarrolladores de esa implementación [9], inicialmente no se esperaba que se llegaran a soluciones de tal calidad en tan poco tiempo usando estas heurísticas.

### 2.3.6 Depth First Search

En el año 2009, David C. Uthus, Patricia J. Riddle y Hans W. Guesgen [14] proponen resolver TTP usando un método basado Depth First Search (DFS).

Usualmente se habla de DFS como un algritmo 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 |

Figure 8: Resultados del autómata de aprendizaje en las instancias de National League

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

# 3 Técnica Propuesta - Greedy

A continuación, se propone una implementación basada en algoritmos greedy.

Un algoritmo greedy es un algoritmo constructivo que, paso a paso, decide según una función miope que componentes agregar a la solución actual. Los algoritmos greedy son increíblemente simples y fáciles de implementar, aunque las soluciones obtenidas a partir de estos no suelen ser las óptimas. Sin embargo, al adentrarse en la búsqueda del método de solución a un problema, los algoritmos greedy son, en la mayoría de los casos, un excelente punto de partida, que en el futuro se puede extender o modificar usando heurísticas más complejas.

Existen 4 cosas que hay que definir para cada algoritmo greedy:

1. **Representación:** en que formato se representará la solución (incompleta o completa al final) actual, paso a paso.
2. **Punto de partida:** solución inicial, incompleta por definición, desde la cual se completará paso a paso según la función miope.
3. **Función miope:** que cambios en la solución actual se realizan en cada paso y bajo que criterio se escogen los cambios.
4. **Función de evaluación:** una vez obtenida la solución completa y factible,la función de evaluación indica la calidad de esta, según el problema.

A continuación se explican estas componentes para la implementación:

## 3.1 Representación

La representación usada para esta implementación será la misma mencionada en la sección 3.2 [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 $k$.

## 3.2 Punto de partida

La idea de este algoritmo es construir una solución factible desde una solución no factible, es decir, que no cumpla con todas las restricciones. Dado que una solución no factible no puede servir como solución final entregada, si una solución no es factible, no es considerada como solución. Por lo tanto, reparar una solución no factible, transformándola en factible podría considerarse como un algoritmo constructivo y, por lo tanto, se puede aplicar greedy en este caso.

Teniendo en cuenta esto, el punto de partida de nuestro greedy es un torneo Double Round Robin ya programado, el cumple con las restricciones intrínsecas de DRR (cada equipo juega una y solo una vez por round, cada equipo juega una vez de local contra todo el resto de equipos y cada equipo juega una vez de visita contra todo el resto de equipos), pero viola las restricciones que añade TTP (atmost y norepeat). Adelantándonos a la función miope, el algoritmo greedy consiste en reducir a cero la cantidad de violaciones existentes de este torneo inicial.

Por lo tanto, para obtener una solución inicial es necesario generar un torneo DRR. Existen algoritmos muy simples para crear torneos RR, los cuales se pueden extender a DRR al duplicar sus rondas, invirtiendo sus partidos (invirtiendo los roles local-visita)

Round Robin:

| | r1 | r2 | r3 |
|---|---|---|---|
| T1 | T2 | T3 | T4 |
| T2 | -T1 | T4 | T3 |
| T3 | T4 | -T1 | -T2 |
| T4 | -T3 | -T2 | -T1 |

Double Round Robin (duplicado)

| | r1 | r2 | r3 | r4 | r5 | r6 |
|---|---|---|---|---|---|---|
| T1 | T2 | T3 | T4 | -T2 | -T3 | -T4 |
| T2 | -T1 | T4 | T3 | T1 | -T4 | -T3 |
| T3 | T4 | -T1 | -T2 | -T4 | T1 | T2 |
| T4 | -T3 | -T2 | -T1 | T3 | T2 | T1 |

Figure 9: Ejemplo de duplicación de un torneo RR a DRR, con $N = 4$. Es importante tomar en cuenta que, a diferencia de lo que se ve en este ejemplo, las rondas $N-1$ "duplicadas" no necesariamente tienen que ir justo después de las $N-1$ rondas originales, sino que también se pueden mezclar entre ellas.

Para esta implementación se utilizará el "algoritmo de rotación" para generar torneos RR, modificado para construir partidos DRR aleatorios.

El algoritmo de rotación consiste en generar varios emparejamientos de equipos (de $N/2$ partidos en cada emparejamiento) tales que al insertarlos uno en cada ronda, se genere un torneo RR factible.

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


Figure 10: Ejemplo de $E_0$. En este caso, los equipos de arriba jugarían de local y los de abajo de visita. El equipo 1 (L) juega contra el 4 (V), el 2 (L) contra el 5 (V) y el 3 (L) contra el 6(V)

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

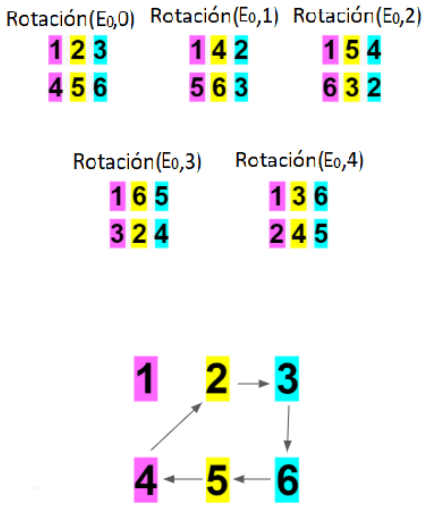


Figure 11: Ejemplo de rotaciones posibles de $E_0$, según distintos valores de $p$. La forma en que se rotan los equipos de describe abajo. Nótese que: rotación$(E_0, 0) = E_0$

.

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

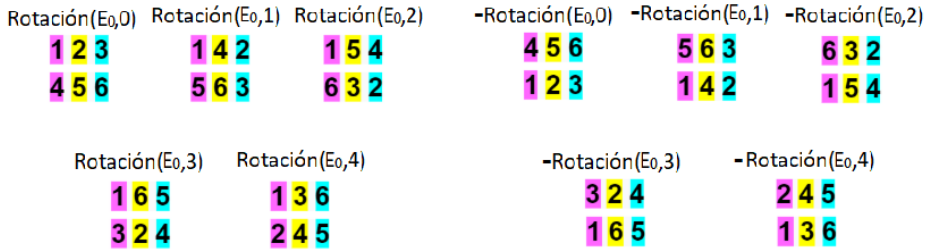


Figure 12: Ejemplo de rotaciones posibles de $E_0$ (a la izquierda), y sus respectivos inversos (a la derecha), en los cuales se invierten los roles de local (tríos de "arriba") y visita (tríos de "abajo")

.

Si se llenan las $2*(N-1)$ rondas con estos emparejamientos (aleatoriamente), queda una solución de DRR factible (aunque muy probablemente NO sea una solución factible de TTP). Por lo tanto, el algoritmo de generación de puntos de partida para nuestro greedy es, en pseudocódigo:

---

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

---

## 3.3 Función Miope

En cada iteración se le aplicará al torneo actual $S$ la transformación que minimice el valor:

$$totalConflicts(S) = |conflictedRounds(S)| + |conflictedTeams(S)|$$

Donde $conflictedRounds(S)$ y $conflictedteams(S)$ son respectivamente los conjuntos de las rondas y equipos en donde se encuentren infracciones a las restricciones que TTP añade: atmost y norepeat (el resto de restricciones ya se cumplen por ser un DRR) .

En caso de que un equipo $g > 3$ partidos consecutivos como local o más de 3 partidos consecutivos como visita (violando la restricción atmost), ese equipo será parte del conjunto $conflictedTeams(S)$ y las $g$ rondas en que se están jugando esos $g$ partidos serán parte de $conflictedRound(S)$. Por otro lado, si dos equipos juegan entre sí en dos rondas consecutivas (violando la restricción norepeat), los dos equipos y las dos rondas serán parte de $conflictedTeams(S)$ y $conflictedRound(S)$, respectivamente.

Es necesario mencionar que un equipo o una ronda no puede aparecer más de una vez en $conflictedTeams(S)$ y $conflictedRound(S)$ respectivamente. Si dentro de una ronda, por ejemplo, dos equipos están violando la restricción atmost, la ronda solo aparecerá una vez en $conflictedRound(S)$.

Con respecto a las transformaciones, usaremos dos de las que se implementaron en la sección 2.3.2 [1]: $SwapTeams(T_i, T_j)$ y $SwapRounds(r_i, r_j)$. La

razón de porqué se escogieron estas transformaciones es la simplicidad y facilidad de implementación de ellas.

Cada movimiento posible debe involucrar a por lo menos uno de los equipos pertenecientes a $conflictedTeams(S)$ (en el caso de $SwapTeams$) o a una de las rondas pertenecientes a $conflictedRounds(S)$ (en el caso de $SwapRounds$), puesto que de nada sirve realizar un cambio de equipos o rondas si este no va a incidir sobre los de conflictos (por ejemplo, $SwapTeams(3, 4)$ no es un movimiento disponible si ni el equipo 3 ni el equipo 4 se encuentran e $conflictedTeams(S)$).

Por lo tanto, en cada iteración, se escoge el movimiento disponible que minimice $totalConflicts(S)$, estos movimientos pueden ser:

- $SwapRounds(r_i, r_j)$, $\forall$ $i$,$j \in [0, 2(N-1)-1]$,
  con $r_i \in conflictedRounds(S)$, o $r_j \in conflictedRounds(S)$.
- $SwapTeams(T_i, T_j)$, $\forall$ $i$,$j \in [1, N]$,
  con $T_i \in conflictedTeams(S)$, o $T_j \in conflictedTeams(S)$.

En caso de que existan varios movimientos que minimicen $conflictedRounds(S)$, se escoge el movimiento que minimice la distancia total de viaje.

Se repite este proceso hasta que $totalConflicts(S) = 0$, es decir, se satisfagan todas las restricciones. En este momento, se obtendría una solución factible. También el algoritmo se detiene en caso de que no hayan movimientos disponibles o movimientos que reduzcan $totalConflicts$. Si en ese caso $totalConflicts$ sigue siendo mayor a 0, el algoritmo habrá fallado en generar una solución factible.
$(\forall i, j)$

## 3.4 Función de evaluación

La función de evaluación es la misma aplicable para una solución factible generada por cualquier algoritmo que resuelva TTP: distancia total de viaje.

Es necesario destacar que, aunque en parte este algoritmo reduzca la función de evaluación durante las iteraciones (en el caso de que se tenga un "empate" entre movimientos que reduzcan la misma cantidad de $totalConflicts$), no se busca optimizar esta función de evaluación ni equipararse con los resultados obtenidos por las implementaciones presentadas en la sección 2. El objetivo es partir desde lo básico (considerando que greedy search es una heurística simple), para familiarizarse con el entorno del problema y escalar desde ahí a algoritmos más complejos y más eficaces.

# 4 Escenario Experimental

Se probará la implementación del algoritmo explicado en la sección 3, echa en C++, desde un sistema operativo Ubuntu 20.04, corriendo con un procesador intel core i3-6006 CPU @ 2.00Hz.

Se utilizarán las instancias del benchmark "national leauge" [6], de $N = 4$, $N = 6$, $N = 10$, $N = 12$ y $N = 14$.

En cuanto al experimento, se ejecutará el algoritmo 1000 veces para cada instancia. De estas, se medirá la cantidad de éxitos en la creación de soluciones factibles en relación a las soluciones totales (pues no se descarta la posibilidad de que el algoritmo entregue soluciones infactibles), la mejor función de evaluación obtenida dentro de las soluciones factibles ($F_{min}$ : menor distancia total de viajes), la peor función de evaluación obtenida dentro de las soluciones factibles ($F_{max}$ : mayor distancia total de viajes) y la función de evaluación promedio de todas las soluciones factibles ($\overline{F}$ ).

# 5 Resultados obtenidos

Probando la implementación sobre las distintas instancias, de acuerdo a lo establecido en la sección 4, se obtuvieron los siguientes resultados:

| 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 |

Lo primero que hay que destacar es que en cada implementación, se obtuvo un 100% de éxito en la creación de soluciones factibles de los 1000 intentos por iteración. Esto quiere decir que el algoritmo prácticamente garantiza la creación de una solución factible.

Comparando estos resultados con los obtenidos por la implementación del LA [9] (muchos de los cuales llegaron a ser óptimos), se llega a la conclusión de que a medida que aumenta $N$, menor es la probabilidad de que la mejor solución obtenida por el algoritmo greedy se asemeje a la óptima.

Por ejemplo, cuando $N = 4$ y $N = 6$, de las mil soluciones generadas en cada instancia, hubieron algunas cuyas las funciones de evaluación de las mejores se acercaron a las funciones de evaluación obtenidas por el autómata de aprendizaje (8559 se acercó a 8276 y 26256 se acercó a 23916, respectivamente).

En cambio, para valores grandes de $N$, las mejor funciones de evaluación obtenidas se alejan cada vez mas de las obtenidas por el LA. En $N = 10$, la

mejor solución obtenida en greedy (79454) es 1.3 veces (aprox) "menos mejor" que la mejor obtenida por LA (59583) y en $N = 16$, la mejor solución obtenida en greedy (409915) es casi 1.47 "menos mejor" que la mejor obtenida por LA (279330). Además, hay que considerar que estamos comparando a tan solo las mejores soluciones de las 1000 obtenidas en cada implementación.

En relación al tiempo de ejecución, aumenta considerablemente en relación a $N$. Para $N = 14$ el algoritmo demora poco menos de un minuto y para $N = 16$, se acerca a los 2 minutos (120 segundos). Entre estas dos instancias, el tiempo de ejecución aumenta en el doble (52 a 104 segundos).

Curiosamente, el aumento de tiempo es mucho mayor entre instancias de valores de $N$ pequeños, por ejemplo, entre $N = 4$ (0.05 segundos) y $N = 6$ (0.55 segundos), el tiempo aumentó en 11 veces su valor, razón mucho mayor a la presente cuando $N$ toma valores mayores, como de $N = 14$ a $N = 16$ (en la cual la razón sería d 2).

Una razón de porque el algoritmo aumenta tanto su tiempo de ejecución en relación a $N$ es el hecho de que para cada movimiento posible en cada iteración hay que calcular las filas en conflicto, las columnas en conflicto y la distancia total de viaje del torneo que se generaría lo que implica revisar todo el torneo. Esto conlleva una complejidad de $O(N^2)$ en relación al tiempo en cada evaluación.

Quizás el mayor incremento de tiempo entre instancias de $N$ más pequeñas se deba a que, a medida que aumenta el número de equipo, se reduce la probabilidad de que hayan muchos conflictos puesto a que el tamaño del torneo (tanto en equipos como en rondas) permite combinaciones menos conflictivas al variar más los equipos. De todas formas, el hecho de que las revisiones en sí aumentan de tiempo en proporción a $N$ hace que las instancias de $N$ mayor demoren mucho más.

# 6 Conclusiones

Como ha sido establecido a lo largo de este informe, greedy search es una heurística que destaca por su simpleza mas no necesariamente por su eficacia. Al tratarse de algoritmos constructivos, varias implementaciones basadas en heurísticas reparadoras como GA y SA que se centran en mejorar soluciones iniciales pueden usar greedy para crearlas.

En muchos casos greedy es un algoritmo rápido y práctico puesto que se construye una solución componente por componente y las funciones miope suelen (aunque no siempre) revisar el "componente candidato" en lugar de toda la "solución candidata" que se formaría al agregar este.

Por ejemplo, en el clásico problema TSP (travelling salesman problem), en donde un "viajero" debe recorrer varias ciudades minimizando su distancia de viaje, un algoritmo greedy consiste en que, desde una ciudad inicial, el viajero

se mueva a la ciudad más cercana de la ciudad en que el está, de las que aún no ha visitado. Es un algoritmo eficiente porque la función miope revisa cual es la ciudad más cercana a la actual, no cual sería la distancia total de viaje actual si el viajero se mueve a cada ciudad.

Sin embargo, la función miope del algoritmo greedy propuesto en este informe (la implementación) exige revisar todo el torneo por cada transformación disponible. Por lo tanto, si se quiere usar este algoritmo como un método de generación de soluciones iniciales para algoritmos más complejos, debería tomarse una de las siguientes medidas:

1. Modificar la implementación de este algoritmo tal que en lugar de revisar la cantidad de conflictos por movimiento en todo el torneo, solo se revisen las rondas y los equipos en los que podrían ocurrir violaciones a las restricciones en caso de que se realice la transformación.

2. Usar un método de generación de soluciones iniciales similar al propuesto, pero el lugar de estar centrado en generar soluciones totalmente factibles, relaje en parte las restricciones de TTP permitiendo en el futuro buscar una solución óptima al explorar por el espacio de las soluciones infactibles (aunque solo acepte soluciones factibles como finales).

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

[4] 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".

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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".

[14] 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.