PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO
FACULTAD DE INGENIERIA
ESCUELA DE INGENIERIA INFORMATICA

ALGORITMOS MEMETICOS Y SU APLICACION EN FIXTURES DEPORTIVOS

JOHN PAUL, BARRIL ARENAS

MEMORIA PARA OPTAR
AL TITULO PROFESIONAL DE
INGENIERO CIVIL EN INFORMATICA

DICIEMBRE 2005

PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO
FACULTAD DE INGENIERIA
ESCUELA DE INGENIERIA INFORMATICA

ACTA DE APROBACION

JOHN PAUL, BARRIL ARENAS

BRODERICK CRAWFORD LABRIN
PROFESOR GUIA

DICIEMBRE 2005

Dedicado a mis padres por todo el amor y preocupación constante.

A mis hermanos.

John

AGRADECIMIENTOS

En primer lugar quiero agradecer a Dios por sus cuidados durante todos estos años, especialmente aquellos lejos de casa.

Deseo agradecer a mi familia por la confianza sin límites y la tranquilidad ofrecida para poder desempeñarme de buena forma en la búsqueda de mis sueños.

Finalmente, quiero dar las gracias a la Universidad por formarme como profesional y entregarme valores que han servido para crecer como persona.

RESUMEN

El presente trabajo de titulación documenta un estudio asociado con el tema de Metaheurísticas, las cuales son procesos maestros de búsqueda que ayudan a dar respuestas de buena calidad a problemas donde la cantidad de posibles soluciones es extremadamente grande.

Se presenta la aplicación de una Metaheurística llamada *Algoritmos Meméticos (AM),* en un problema general denominado *Mirrored Traveling Tournament Problem (MTTP),* que consiste en la minimización de los kilómetros que deben recorrer los equipos involucrados en un torneo deportivo para terminar de enfrentarse. Además se comparan los resultados obtenidos con los mejores hasta el momento, para el problema planteado.

La obtención de resultados se realiza a través de la programación en lenguaje C, de todas las etapas definidas por la Metaheurística y la utilización de los datos entregados por la definición del problema general, en este caso, las distancias de la liga mayor de baseball de Estados Unidos y Canadá.

ABSTRACT

The present work of degree documents a study associated with the subject of Metaheuristics, which are masterful search processes that help giving good quality answers to problems where the amount of possible solutions is extremely great.

The application of a Metaheurístic called *Memetic Algorithms* (*MA*) is presented, in a general problem denominated *Mirrored Traveling Tournament Problem* (*MTTP*), that consists of the minimization of the kilometers that must travel the involved teams in a sport match to finish facing each other. Furthermore the obtained results are compared with the best ones until the moment, for the outlined problem.

The results obtaining is made through the C language programming of all the stages defined by the Metaheuristic and the use of the data given by the definition of the general problem, in this case, the distances of the mayor baseball league of the United States and Canada.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

LISTA DE TERMINOS

AM	Algoritmo Memético
TTP	Traveling Tournament Problem
MTTP	Mirrored Traveling Tournament Problem
NP	Non Polinomial
TSP	Traveling Salesman Problem
AE	Algoritmos Evolutivos
AMAG	Algoritmo Memético Autogenerado
SA	Simulated Annealing
SRR	Single Round Robin
DRR	Double Round Robin
MDRR	Mirrored Double Round Robin
CSP	Constraint Satisfaction Problem
MLB	Mayor League Baseball
NL	National League
IPR	Intercambio Parcial de Rondas
IR	Intercambio de Rondas
IE	Intercambio de Equipo
HC	Hill Climbing
IL	Intercambio de Localías

INTRODUCCION

El avance del tiempo y el exponencial desarrollo de la ciencia de la computación nos permiten observar como problemas de alta complejidad pueden ser resueltos de forma óptima y eficiente, todo lo cual se concreta con novedosos instrumentos y métodos, frutos de una poderosa era tecnológica.

Los problemas combinatorios son aquellos en donde la solución es un subconjunto de un conjunto total de soluciones. Para resolver esto, se han desarrollado algoritmos que pueden clasificarse como exactos o aproximados. Los primeros tratan de encontrar la solución óptima que resuelva el problema en un tiempo acotado, los segundos sacrifican optimalidad por encontrar una solución buena con un consumo razonable de recursos.

Los algoritmos aproximados se ocupan de resolver los problemas llamados “Duros”. Un problema “Duro”, es aquel que generalmente se obtiene al tratar de resolver un caso tomado directamente de la realidad, y se caracteriza por no garantizar que se encuentre una solución en un tiempo razonable.

Entre los algoritmos aproximados, se encuentran los basados en métodos constructivos y otros de búsqueda local. En los últimos años, a los métodos básicos de resolución se han agregados técnicas más generales llamadas Metaheurísticas.

Dentro de las Metaheurísticas encontramos un grupo que constituye el objeto de estudio de la presente investigación. Son las llamadas Metaheurísticas Evolutivas, las que se basan en procesos biológicos como la evolución de las especies para resolver problemas de optimización. Algunos ejemplos de estas técnicas son los Algoritmos Genéticos (AG) y los Algoritmos Meméticos (AM) [2].

Los Algoritmos Meméticos se basan en la idea de evolución cultural de Lamarck. Un *meme* puede ser visto como una idea y es la unidad básica de evolución cultural. Estos algoritmos, conocidos también como Algoritmos Genéticos Híbridos o de Búsqueda Genética Local, nacen de la implementación de la evolución cultural en la técnica, en otros términos, ocurre que los

individuos de una población que en los AG representan las soluciones del problema, en los AM no solo son soluciones, sino que son óptimos locales resultantes de la aplicación de optimizadores a vecindades de la población. Esto acelera la búsqueda y aumenta la calidad de los resultados obtenidos [10].

El diseño de un AM debe solucionar un determinado problema de optimización, en este caso el *Mirrored Traveling Tournament Problem*, La esencia del problema se encuentra en minimizar una función llamada objetivo que representa los kilómetros que deben recorrer los equipos que juegan en un torneo deportivo, para enfrentarse entre sí en una temporada. La restricción que diferencia este problema del original *Traveling Tournament Problem* es que trata con Fixtures **espejados**, esto aumenta la complejidad de encontrar una solución. Existen muchas instancias del problema para ser estudiadas, dependerá del tipo de torneo que se tenga [14].

Los beneficios que tiene realizar este tipo de trabajos se relacionan con hacer que los recursos que se tienen sean utilizados de la forma más eficiente posible, y que los costos asociados se minimicen con el fin de redistribuir los recursos liberados en pos de mejorar la situación existente.

En el caso del Mirrored Traveling Tournament Problem, la minimización de los kilómetros recorridos hace que los costos de viaje de los equipos bajen. Además el tiempo que antes se gastaba viajando, se puede ocupar para el descanso de los jugadores o para tener más sesiones de entrenamiento, útiles para los enfrentamientos del torneo.

OBJETIVOS

OBJETIVOS GLOBALES

- Implementar un Algoritmo Memético aplicado al *Mirrored Traveling Tournament Problem*, que genere soluciones que se acerquen al óptimo en un tiempo razonable de procesamiento.
- Realizar un estudio sobre el algoritmo, documentando el comportamiento de éste para valores diferentes en sus parámetros.

OBJETIVOS ESPECIFICOS

- Modelar el problema de optimización del *Mirrored Traveling Tournament Problem*.
- Diseñar un Algoritmo Memético para resolver el *Mirrored Traveling Tournament Problem*.
- Codificar el Algoritmo Memético en lenguaje C.
- Realizar Pruebas a los diferentes componentes del Algoritmo (Heurísticas).
- Analizar el comportamiento de cada componente del algoritmo ante el cambio en los parámetros correspondientes.
- Realizar códigos alternativos para mejorar el rendimiento del algoritmo originalmente planteado.

ORGANIZACION DEL DOCUMENTO

Este documento se encuentra organizado en capítulos, los cuales se diferencian claramente por la profundidad con la que trata el tema, siendo cada una de estas partes un aporte fundamental para llegar a un resultado que cumpla con los objetivos ya planteados.

El primer capítulo explica que son las Metaheurísticas, se definen sus principales características, se da una clasificación de ellas según su base teórica y métodos de búsqueda. Además se entregan algunas propiedades que deberían cumplir para considerarse beneficiosa.

El segundo capítulo nos introduce a los algoritmos evolutivos, describiendo su historia como base para la definición de los Algoritmos Meméticos (AM). Luego se explican las características principales de los AM, explicando sus partes y la función que estas cumplen dentro del algoritmo.

El tercer capítulo define el problema del armado de fixture deportivos. Comienza generalizando el problema, entregando las opciones que se podrían tomar para armar un buen fixture. Luego se explica el problema formal llamado *Mirrored Traveling Tournament Problem (MTTP)*, sus objetivos y restricciones.

El cuarto capítulo muestra el diseño de la solución siguiendo lo definido en el capítulo 2, explica cada parte del AM aplicado al MTTP. Lo anterior termina en la implementación mostrada en el capítulo 5.

El sexto capítulo muestra los resultados de la implementación y un análisis del algoritmo propuesto. Lo anterior se traduce en las conclusiones para cada prueba realizada.

Finalmente se presentan anexos, que muestran código fuente relevante y algunas ejecuciones del algoritmo implementado en lenguaje C.

CAPITULO 1. METAHEURISTICAS

1.1.- COMPLEJIDAD COMPUTACIONAL

Un problema de optimización se resume en hallar un valor máximo o mínimo de una función dada, la cual es la representación general del problema. Las teorías clásicas de optimización como el cálculo diferencial, tratan el caso en que el dominio del problema, es decir, las posibles instancias de entrada de éste son infinitas.

Los problemas de optimización combinatoria generalmente tienen un dominio finito de posibles soluciones, y puede parecer que resolver este tipo de problemas es más sencillo que el caso anterior, ya que se puede buscar solución por solución la más conveniente. En otras palabras, se podrían generar todas las soluciones posibles, evaluarlas según el costo asociado que tengan y obtener la mejor solución. Sin embargo, aunque un método así en teoría siempre llega a la solución óptima, es demasiado costoso desde el punto de vista de la eficiencia, pues el tiempo que se utilizará en calcular y evaluar las posibles soluciones crece de manera exponencial al aumentar las entradas del problema.

Existen problemas combinatorios para los que no se conocen algoritmos que los resuelvan, además provocan que el tiempo de cálculo crezca exponencialmente cuando crece el tamaño del problema. A estos problemas se les llama difíciles de responder. Por otro lado existen problemas que cuentan con algoritmos que los resuelven de forma eficiente, ya que el tiempo de cómputo crece de forma polinomial cuando aumenta el tamaño del problema.

Se clasificó matemáticamente a los problemas de optimización según el algoritmo que encuentra el óptimo para dicho problema, y el tiempo de cómputo que consumen para ello. Aquellos problemas que cuentan con un algoritmo que los resuelve en un tiempo polinomial, se dicen que pertenece a la clase P, y se considera resuelto de forma eficiente. Sin embargo, se ha probado que la mayoría de los problemas de optimización pertenecen a un superconjunto de la clase P, denominada NP, en la cual se encuentran aquellos problemas de los cuales no se conocen algoritmos que los resuelvan en tiempo polinomial. Algunos problemas que caen en esta clasificación son:

- Problema del vendedor viajero (TSP).
- Coloreo de Grafos
- Traveling Tournament Problem

Todos los problemas que pertenecen a la clase NP pueden ser reducidos polinomialmente a todos los demás, es decir si existe una solución en tiempo polinomial para uno de ellos, se podría dar para todos los problemas de NP.

Se dice que un problema es NP-duro, si cualquier problema en NP es polinomialmente transformable en él, aunque el problema en sí no pertenezca a NP. Si el problema además pertenece a NP, se lo denomina NP-completo.

Hasta ahora no se ha podido encontrar algoritmos eficientes para problemas NP-completos [1].

1.2.- METAHEURISTICAS

En Inteligencia Artificial se emplea el calificativo de Heurístico en un sentido muy genérico, para aplicarlo a todos aquellos aspectos que tienen que ver con el empleo de conocimiento en la realización dinámica de tareas [23].

Dado la dificultad que existe al momento de resolver un problema combinatorio, comenzaron a aparecer técnicas o procedimientos que encontraban soluciones, si bien no óptimas, factibles de implementar, es decir, su calidad era suficientemente buena y se lograba en un tiempo de procesamiento razonable.

La palabra griega “Heuriskein” que significa “Encontrar”, dio origen al término Heurística, que en general consiste en un conjunto de pasos definidos para guiar las tareas a realizar para encontrar la solución a un problema dado o satisfacer un conjunto de restricciones, también son llamados “Algoritmos de Aproximación”.

Si bien los estudios sobre Heurísticas son cada vez más avanzados, es bueno recalcar que son procesos que se basan en el sentido común, es decir, siguen un patrón lógico y se potencian con la capacidad de procesamiento de los equipos actuales de computación [2].

Existen algunos factores que pueden producir que utilizar Heurísticas para la resolución de un problema, sea atractivo [3]:

a) Cuando no se puede encontrar un método exacto de resolución.
b) Cuando encontrar una solución óptima no es necesario, sólo basta con encontrar una buena. Las razones de esto pueden ser, el alto costo que generaría buscar el óptimo y el largo tiempo de espera para ello.
c) Cuando no se cuenta con datos seguros o fidedignos, es decir, el problema era tan complejo que se tuvo que simplificar el modelo original.
d) Cuando no se cuenta con el equipo computacional indicado para procesamientos complejos. Se beneficiará la respuesta rápida a costa de precisión.

Una de las ventajas de utilizar Heurísticas es que pueden generar más de una solución, así se puede flexibilizar la utilización de una u otra dependiendo de cual se adapta mejor a la realidad de quien la quiera. Esto puede ocurrir cuando no se han podido modelar restricciones o bien se ha simplificado el modelo original.

Existe una clasificación de Heurísticas, dependiendo el modo en que buscan y construyen sus soluciones, es así como se separan en [23]:

a) Métodos Constructivos: añaden paulatinamente componentes individuales a la solución, hasta que se obtiene una solución factible.
b) Métodos de mejora local: parten de una solución factible y mediante operaciones sobre la solución, se llega a mejorarla hasta que se cumpla un determinado criterio de parada.

El término Metaheurística quiere decir “más allá de la Heurística”, y formalmente es un proceso maestro iterativo, que guía y modifica operaciones Heurísticas subordinadas para

producir, de forma eficiente, soluciones de alta calidad. En cada iteración puede manipular una solución o un conjunto de soluciones. Las Heurísticas subordinadas pueden ser procedimientos de alto o bajo nivel, simplemente una búsqueda local o un método constructivo [4].

1.3.- CLASIFICACION DE METAHEURISTICAS

Como las Metaheurísticas son estrategias para diseñar heurísticas, se pueden dividir en tipos según el procedimiento al que se refiere, es decir, la forma en que se quiere lograr una solución de buena calidad. Algunos de los tipos fundamentales de Metaheurísticas son los de Relajación, Constructivas, de Búsqueda Local y Evolutivas [2].

a) *Metaheurística de Relajación*: al momento de encarar un problema del mundo real, es necesario modelar dicho problema. Puede que el modelo resultante tenga una complejidad tal que haga difícil encontrar una solución buena con un costo bajo en recursos. Así, un método que simplifique el modelo es la alternativa que se puede tomar, es decir, que relaje el modelo inicial haciendo que las restricciones duras se debiliten o bien desaparezcan, haciendo el problema más fácil de resolver. Las Metaheurísticas de relajación son estrategias para el empleo de simplificaciones del problema en el diseño de la heurística (formulaciones del problema y soluciones).

   Algunos ejemplos de Metaheurísticas de Relajación son los métodos de relajación Lagrangiana o subordinadas. En resumen, una solución de este tipo de método no es tan exacto u óptimo pero, en general, se gana en tiempo de procesamiento.

b) *Metaheurísticas Constructivas:* estos métodos aportan soluciones del problema por medio de un procedimiento que incorpora iterativamente elementos a una estructura inicialmente vacía, que representa a una solución. Las Metaheurísticas Constructivas establecen estrategias para seleccionar las componentes con las que se construye una buena solución al problema. Entre las Metaheurísticas más conocidas de este tipo se encuentran GRASP.

c) *Metaheurísticas de Búsqueda Local:* este es el tipo de Metaheurística más importante. La estrategia que utilizan es la de recorrer el espacio de soluciones factibles. La representación de las soluciones se realiza a través de una codificación que incluya toda la información necesaria para su identificación y evaluación. Una búsqueda sobre el espacio consiste en generar una sucesión de puntos, pasando de uno a otro por medio de una serie de transformaciones o movimientos. Las Metaheurísticas de búsqueda proporcionan pautas para obtener recorridos que, con alto rendimiento proporcionen soluciones de alta calidad.

d) *Metaheurísticas Evolutivas:* estas Metaheurísticas establecen estrategias para conducir la evolución en el espacio de búsqueda de un conjunto de soluciones (usualmente llamados Población), con la intención de acercarse a la solución óptima con sus elementos.

   Existen variados algoritmos evolutivos, los que se distinguen por la forma en que combinan la información proporcionada por los elementos de la población para hacerla evolucionar mediante la obtención de nuevas soluciones. Entre los algoritmos más conocidos se encuentran, Algoritmos Genéticos, Estimación de Distribuciones y la técnica que se estudiará en este informe, Algoritmos Meméticos.

e) *Otros:* aquí podemos encontrar Metaheurísticas que contienen características de varios de los tipos antes descritos. Entre ellas se encuentran las Metaheurísticas de descomposición y las de memoria a largo plazo. Las primeras descomponen el problema en subproblemas y los resuelven por separado. Las de memoria a largo plazo son heurísticas de aprendizaje, y utilizan la información de las soluciones de alta calidad que se generan para ajustar los criterios de selección futura.

1.4.- PROPIEDADES PARA UNA BUENA METAHEURISTICA

Las Metaheurísticas en general, deben cumplir con las propiedades siguientes [2]:

- *Simple* : cualquier persona utiliza algún principio de Metaheurística en su vida, esto hace que algunas más que otras sean fáciles de entender en su esencia. Esta propiedad sugiere que el sentido común ayuda a esclarecer la idea central de la Metaheurística.

- *Precisa* : las fases de la Metaheurística deben estar definidas de forma clara, y formulada de manera tal que cada una por separado sea una tarea concreta. Esta propiedad ayuda en el caso de utilizar la Metaheurística en distintos tipos de problemas.

- *Coherente* : los elementos de la Metaheurística deben ser posibles identificarlos de manera natural. Esto es, conociendo el principio que las rige poder conocer la forma en que se compone, y un poco de su comportamiento.

- *Efectiva* : los algoritmos derivados de la Metaheurística, deben cumplir con la misión de encontrar soluciones de alta calidad, pudiendo éstas ser o no óptimos para ese problema.

- *Eficaz* : la probabilidad de encontrar una solución de alta calidad con la Metaheurística debe ser alta. Esto depende del problema y la elección del equipo de trabajo.

- *Eficiente* : la Metaheurística elegida debe hacer un buen uso de los recursos con los que se cuente.

- *General* : la Metaheurística debe poder ser aplicada a una variedad de problemas, no afectando en demasía su buen rendimiento.

- *Adaptable* : la Metaheurística debe ser capaz de adaptarse a distintos contextos de aplicación como a cambios en el modelo inicialmente construido.

- *Interactiva* : se debe permitir que el usuario pueda aplicar sus conocimientos para mejorar el rendimiento del procedimiento.

- *Múltiple* : la Metaheurística debe tener la capacidad de entregar varias soluciones de alta calidad entre las que el usuario pueda elegir.

CAPITULO 2. ALGORITMOS MEMETICOS

2.1.- HISTORIA DE LOS ALGORITMOS EVOLUTIVOS

En el siglo XIX, Darwin (1809-1882) de la misma forma que Lamarck (1744-1829), observaron la adaptación y la adecuación entre el desempeño de dos seres y su manera de vivir. La diferencia que Darwin presentó en sus investigaciones fue un mecanismo de evolución basado en la selección natural. En tanto, Lamarck defendía un mecanismo de evolución a través de herencia, donde las características adquiridas directamente durante la vida (también denominada ley de uso y desuso), y los cambios ambientales a través de la vida de un organismo causan cambios estructurales que son transmitidos a sus descendientes.

Otra investigación relacionada, fue la realizada por James Mark Baldwin, el cual propuso un nuevo factor de evolución donde las características adquiridas podrían ser indirectamente transmitidas a los descendientes. El efecto Baldwin espera explicar equilibrios puntuales no explicados, a través de la realización de dos etapas. Primero, la plasticidad fenotípica permite a un individuo adaptarse a una mutación bien sucedida, y así ser más apto y proliferar. La segunda etapa dice que dado un tiempo suficiente, un mecanismo de evolución tiende a transformar un comportamiento aprendido "plástico" a uno "rígido", es decir, un comportamiento aprendido una vez en la primera etapa, tiende a ser instintivo en una segunda etapa [5].

Richard Dawkins presentó la evolución desde una perspectiva diferente, utilizó el término *"meme"* para referirse a una idea. Dawkins elaboró una teoría para estudiar y explorar tres fenómenos que son únicos para una evolución cultural [10].

- El cerebro detecta una regularidad y construye esquemas que un operador basado en conocimiento utiliza para adaptarse.
- La Imitación dice que las ideas se propagan cuando los miembros de una sociedad observan y copian unos a otros.
- Simulación Mental, los individuos pueden imaginar que pasaría si fuese implementado un meme antes de pasar recursos a dicho meme.

Con estos tres operadores se define un mecanismo rudimentario de selección [4].

2.2.- FUNDAMENTOS DE LOS ALGORITMOS EVOLUTIVOS

Los métodos de optimización y busca estocástica basados en modelos de evolución biológica natural, han tenido un creciente interés en las últimas décadas debido a la posibilidad que ofrecen en la resolución de problemas complejos, áreas de optimización y aprendizaje de máquina.

Los Algoritmos Evolutivos (AE), se rigen por principios del mundo biológico y se basan en la teoría de evolución de Darwin. Los AE intentan imitar algunos de los mecanismos evolutivos para resolver problemas que requieren adaptación, búsqueda y optimización [2].

En los AE, los puntos en el espacio de búsqueda de soluciones se modelan a través de individuos que integran un ambiente artificial. Los individuos de esta población son tomados en cada iteración, con el fin de que se relacionen y compitan, y así lograr una evolución en la población. La selección de un individuo de la población depende de una función de aptitud, que consiste generalmente en una función objetivo que se desea maximizar o minimizar para solucionar un problema de optimización determinado [4].

Los AE no requieren de conocimiento de las características del problema de optimización, y no dependen de ciertas propiedades de la función objetivo, sino de la evaluación de la función objetivo de los individuos.

Los AE son útiles para búsquedas globales, donde los métodos determinísticos pueden llevar a máximos o mínimos locales. Por esto los AE son aptos para resolver un amplio espectro de problemas no lineales, discontinuos, discretos, multivariables, entre otros. La *Figura 2.1* ejemplifica de manera gráfica los conceptos de óptimo global y local.

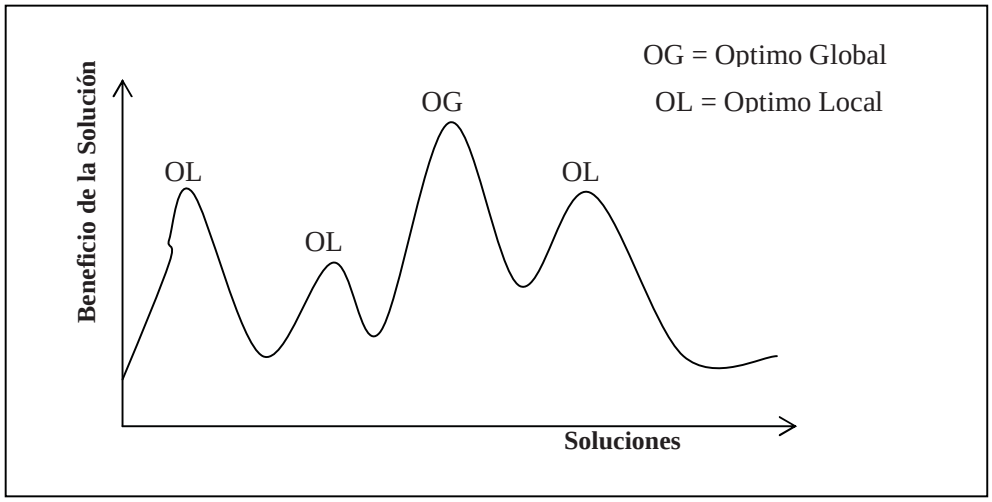


Figura 2.1.- Optimo Global y Local

La estructura básica general de un Algoritmo Evolutivo es la siguiente:

1) Inicialización aleatoria de la población.
2) Evaluación de la función objetivo.
3) Selección de individuos más aptos de acuerdo a la estrategia de selección.
4) Aplicación de operadores de Recombinación y mutación.
5) Generación de una nueva población de soluciones candidatas.
6) Repetición de pasos 2 al 5 hasta satisfacer una condición de parada.

2.2.1.- Algunas Razones Para Utilizar Algoritmos Evolutivos

Las características principales que poseen los AEs son las siguientes [4]:

- Operan sobre población de soluciones
- No requieren cálculos de derivadas e información sobre el gradiente de una función objetivo.
- Presentan simplicidad conceptual
- No son afectados en cuanto a eficiencia, cuando existen discontinuidades y ruidos en determinados problemas.

2.2.2.- Potencialidades

Los AEs poseen una serie de habilidades útiles [4]:

- Tratan adecuadamente sistemas sujetos a restricciones.
- Se adecuan a implementaciones en paralelo y distribuidas.
- Posibilitan la utilización del conocimiento obtenido a priori por el usuario.
- Tratan con espacios de búsqueda complejos.

2.2.3.- Limitaciones

- Puede que para distintas ejecuciones entreguen respuestas diferentes.
- Requiere muchas evaluaciones a la función objetivo.

2.3.- ALGORITMOS MEMETICOS

2.3.1.- Introducción

El origen del término "Algoritmos Meméticos", se remonta a finales de los años ochenta con el trabajo de Pablo Moscato *"On Evolution, Search Optimization, Genetic Algorithm and Martial Arts: Towards Memetic Algorithm"*. Donde se discutía la utilización de *Simulated Annealing* como método de búsqueda local, en el marco de una Heurística competitiva-cooperativa en una población de agentes, aplicado al problema del vendedor viajero (TSP). El nombre nace del término "meme" (definido por Dawkins) que es la unidad mínima de transmisión cultural, la cual puede verse como una idea que viaja de cerebro en cerebro haciendo que los individuos mejoren o evolucionen para un mejor comportamiento en el medio ambiente. Un ejemplo de transmisión cultural es la imitación, es decir, si a un individuo *X* se le presenta un problema *Y*, tarda *T* tiempo en resolverlo y es visto por *Z*, se puede decir que debido a la transmisión cultural, si a *Z* se le presenta el problema *Y*, tardará $T' < T$ tiempo en resolver dicho problema [9].

Un Algoritmo Memético (AM) es una heurística evolutiva del tipo poblacional, es decir, mantiene una población de individuos llamados agentes, que representan soluciones factibles del problema determinado. Es aquí donde aparece una diferencia con el Algoritmo Genético (AG), ya que un agente puede representar más de una solución al problema. Esto por que siguiendo la definición anterior de transmisión cultural, un agente representa una evolución al ser capaz de resolver el problema de manera más eficiente que otras de las cuales aprendió. Computacionalmente, esta evolución se logra a través de la inclusión de optimizadores locales al proceso genético, logrando que los agentes (cromosomas) sean óptimos locales, es decir, tengan una buena calidad para su posterior reproducción, lo que aumentará la probabilidad de encontrar una solución cercana al óptimo global más rápidamente. Los agentes interrelacionan entre sí en un ambiente de cooperación y competición, de manera similar a lo que ocurre en la naturaleza entre individuos de una misma especie, con el fin de realizar un paso generacional y evolucionar la población [10].

El paso generacional supone una mejora en los agentes. Esto se hace mediante operadores biológicos, Recombinación y Mutación, los cuales trabajan en conjunto para crear nuevos individuos (agentes), cuyas características en gran parte serán combinaciones genéticas de sus padres (dos o más agentes de la generación anterior). La Recombinación es el proceso que realiza el cruce de genes entre dos o más agentes llamados padres, para crear un agente nuevo con características comunes, además de esto aplica una estrategia de selección con el fin de realizar el paso generacional. Existen mecanismos definidos para realizar este proceso, los que serán explicados más adelante [11].

El operador de mutación es el encargado de crear un agente mediante la modificación de uno existente, incluyendo conocimiento adquirido generalmente acerca del problema. Existen los llamados meta-operadores de mutación, los cuales son uno de los rasgos más distintivos de los AM. Estos meta-operadores iteran la aplicación del operador de mutación, conservando los rasgos del que obtuvo alguna mejora en la creación del nuevo agente. Computacionalmente se le conoce con el nombre de optimizador local [23,10]. En la *Figura 2.2* se grafica el comportamiento general de un AM [12].

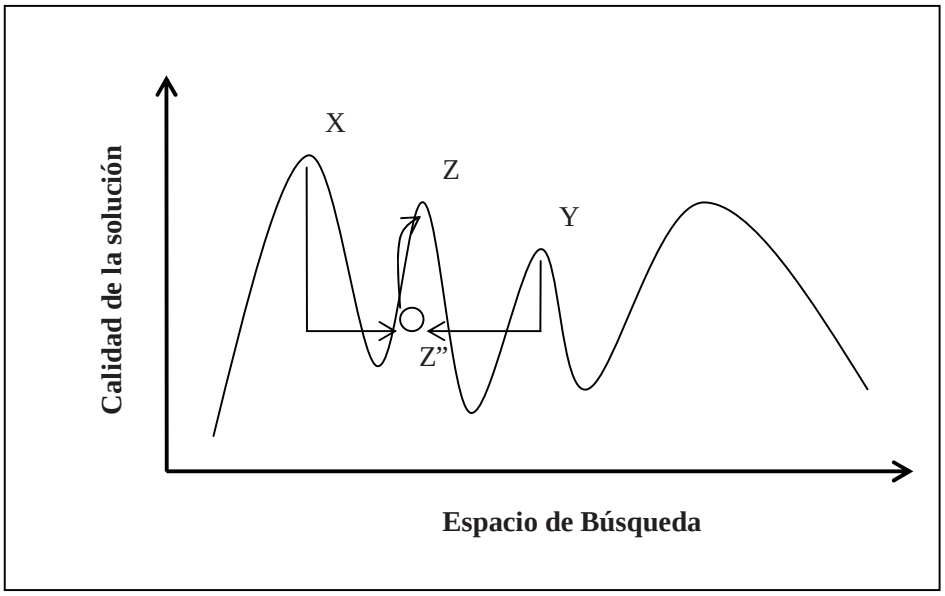


Figura 2.2.- Comportamiento de un AM

El AM busca en el sub-espacio de óptimos locales dentro del espacio de búsqueda total de soluciones. Luego se toma como padres X e Y, se aplica Recombinación creando un hijo Z, el cual es reparado mediante un optimizador local con el fin de buscar un óptimo cercano a él.

Algorítmicamente las características más distintivas de un AM son la incorporación de operadores de búsqueda local y conocimiento del problema. Los operadores de búsqueda local producen una disminución del tiempo en que se entrega una respuesta de calidad, ya que los agentes realizan una búsqueda propia en el espacio total haciendo que la población inicial contenga agentes prometedores, lo que reduce el espacio de búsqueda y acelera el proceso de encontrar una buena solución. El aprovechamiento del conocimiento del problema se logra gracias a la incorporación de operadores de Recombinación inteligentes (propios del problema a resolver). Entonces, al asignar valores a los genes de la descendencia según los valores de los padres, se está aprovechando lo calculado por la aplicación de Recombinación a agentes previos, logrando óptimos locales nuevos a partir de óptimos locales encontrados anteriormente [3].

Con este enfoque se tiende a subsanar la dependencia con respecto a soluciones iniciales. Si bien los descendientes toman la mayoría de los genes de alguno de sus padres, pueden tener algunos elegidos al azar.

Los conceptos básicos para la creación de un AM se explicarán en la S*ección 2.3.2*. Estas definiciones aclararán ciertos conceptos necesarios para un diseño concordante con la inspiración de la técnica. Luego, en la *Sección 2.3.3* se explicarán las diferentes etapas que componen un AM.

Una de las tendencias más importantes de evolución de los AM es la de Algoritmos Meméticos Autogenerados (AMAGs), los que se basan en la buena elección de la Heurística de búsqueda local y cómo los resultados finales son muy dependientes de ella. Se ha demostrado que la elección de la heurística de búsqueda local no depende sólo del problema que se esté tratando, sino que, del momento particular del proceso de búsqueda en el cual se toma la decisión.

Diseñar AMs que se adapten "al vuelo" a condiciones dinámicas es posible. Los AMAGs, al contrario de los AMs tradicionales donde sólo existe un operador de búsqueda local o un conjunto fijo de ellos donde elegir, crea de manera co-evolutiva las estrategias de búsqueda local con las soluciones en donde esas serán aplicadas. De esta manera los AMAGs están más cerca del concepto de "*meme*" de Dawkins visto en la *Sección 2.1* [18].

Actualmente se trabaja en un proyecto llamado "MemePool", el cual propone la orientación a objetos como el paradigma a utilizar para el desarrollo de los AM y a Java como lenguaje de programación. La finalidad de esto es la reutilización de componentes para diferentes problemas [19].

2.3.2.- Definiciones Básicas Para Un Algoritmo Memético

Un problema P tiene un conjunto de entradas denotado por I*p*. Para cada instancia *x* perteneciente a I*p*, existe una solución denotada por *SOL(x)* que representa una solución posible al problema.

Se debe desarrollar un algoritmo que resuelva P, dada la instancia *x*, entregando un elemento *y* del conjunto de respuestas *RESP(x)*, que satisface los requerimientos del problema. En muchos casos se define un booleano *POSIBLE(x,y)*, que representa la relación verdadera o falsa entre la instancia *x*, y la respuesta *y*.

Las condiciones que debe cumplir un problema combinatorio para ser resuelto vía AM son:

- La cardinalidad de *SOL(x)* es finita.
- Cada solución *y*, perteneciente a *SOL(x)* tiene un valor entero asociado *mP(y,x)*, obtenido por la evaluación en una *función objetivo mP*.
- Un criterio es definido para distinguir, entre los valores enteros generados por la función objetivo, cual es el mejor de todos.

La idea central es encontrar una solución *y*, tal que no exista ninguna otra mejor según el criterio aplicado.

Ahora definiremos los conceptos de *Espacio de búsqueda, Vecindad, Función Objetivo y Diversidad.*

a) Espacio de búsqueda (*E*): es un conjunto *E(x)*, cuyos elementos tienen las siguientes propiedades.
   - Cada elemento *e* de *E(x)* representa como mínimo una respuesta en *RESP(x).*
   - Por lo menos un elemento de *E(x)* representa una solución al problema, existiendo por supuesto un óptimo que pertenece a *SOL(x).*

Cada elemento de *E(x)* puede verse como una configuración, la cual se relaciona con un elemento en *RESP(x)* ya que puede que una configuración sea impracticable.

b) Vecindad *(V)*: es la responsable de relacionar las diferentes configuraciones. Se define la Función *V*, que asigna cada elemento *e* de *E* a un conjunto $V(e) \subset E$, como vecindario de *e*, es decir, configuraciones similares. El conjunto *V(e)* se llama vecindad de *e*, y su importancia es que permite la aplicación de optimizadores locales.

Llamamos configuraciones similares a aquellas que siendo objeto de algún movimiento u operación simple, genera otra configuración estrictamente diferente, pero parecida.

c) Función Objetivo *(F)*: para definirla necesitamos un conjunto *C*, cuyos elementos son los valores de mejor calidad, además se necesita un criterio de evaluación ($<_f$) para distinguir cual es el mejor entre los valores de *F*.

La función Objetivo se define como:

F: $E \rightarrow C$; con esto relacionamos los elementos del espacio con los valores de calidad que estamos buscando.

d) Diversidad: se entenderá como el grado de similitud entre los elementos de *SOL(x)*.

Al momento de construir un AM es necesario tener presentes estas definiciones para ligarlas con el diseño del algoritmo futuro [11].

2.3.3.- Estructura Funcional De Un Algoritmo Memético

En general un AM se compone funcionalmente de dos grandes etapas, la primera se refiere a la construcción de la población y las técnicas de optimización que darán origen a los agentes, y la segunda es el algoritmo de evolución, el cual está marcado por un ciclo que controla la convergencia, es decir, que se ha alcanzado una solución que no es posible mejorar. Dentro de este ciclo se aplicarán los operadores biológicos y se hará el paso generacional seleccionando a los mejores agentes para que sobrevivan [10].

2.3.3.1.-Inicialización De La Población

Este proceso consiste en la construcción del conjunto de cromosomas que representan las soluciones del problema. Para la construcción pueden usarse Heurísticas constructivas como GRASP, o se pueden utilizar procesos aleatorios de generación de cromosomas. Es importante mencionar que para cada problema se pueden crear procedimientos particulares que se acomoden mejor a los requerimientos y restricciones involucrados.

Un aspecto importante en la inicialización de la población, es el tamaño que esta tendrá. Una población muy pequeña hará que los cromosomas tiendan a la convergencia rápida, lo que puede producir que el algoritmo se estanque en un óptimo local por la perdida de diversidad, es decir, que a medida que los cromosomas evolucionen comiencen a parecerse genéticamente en demasía entre ellos. Por otro lado, si la población es muy grande aumentará dramáticamente el tiempo de cómputo necesario para encontrar una solución por parte del algoritmo, produciéndose un consumo de recursos innecesario [17].

La *Figura 2.3* muestra un seudo–código de la inicialización de la población de agentes en un AM de manera aleatoria [11].

Figura 2.3.- Inicialización de la Población de Agentes.

```
Procedimiento GenerarPoblacionInicial
Inicio
    Inicializar pop usando PoblacionVacía();
    Parfor j ← 1 to TamañoPoblacion do
            I ← GenerateRandomConfiguration();
            J ← local-search (I);
            InsertInPopulation individual I to pop;
    Endparfor
    Retornar pop
Fin
```

2.3.3.2.- Selección

El proceso de selección se utiliza para determinar a los padres de la próxima generación de soluciones, la que deberá ser de mejor calidad que la actual. En otras palabras, este proceso elige los cromosomas a los que les serán aplicados los procesos evolutivos (Recombinación y Mutación), con el fin de mejorar la población. Para esto existen variados métodos y se basan en la calidad o bondad que las distintas soluciones presentan para el problema.

2.3.3.3.- Optimizador Local

Los Algoritmos Meméticos utilizan búsquedas locales y basadas en población. La búsqueda local comienza con un elemento *e* del espacio *E*, generado de forma aleatoria o constructiva por otro algoritmo. Luego se itera, aplicando algún operador sobre la configuración resultando otra dentro de la vecindad, se evalúa si es preferible esta configuración para mantenerla como "la actual mejor". Luego se sigue iterando hasta que se cumpla algún criterio de terminación.

Generalmente un criterio de terminación consiste en iterar una cantidad de veces suficiente sin encontrar mejora en las configuraciones, es decir, si se cumplen N iteraciones sin resultados mejores se termina de iterar y se llega a un óptimo local. Otras veces se utilizan mecanismos complejos basados en estimación de probabilidad [11].

Figura 2.4.- Estructura de una Búsqueda Local.

```
PROCEDIMIENTO Búsqueda Local(Actual)
Inicio
        Repetir
                Nuevo ← GeneraVecino(actual);
                Si (C(Nuevo)  <f  C(Actual) ) entonces
                        Actual ←  Nuevo;
                Fin Si
        Hasta que CriterioTerminacion();
        Retorna Actual
Fin
```

Donde $<_f$ representa el criterio de evaluación de la calidad de la configuración. Las características de esta búsqueda local es normalmente llamada Hill Climbing.

2.3.3.4- Recombinación

En los AG la Recombinación es realizada en analogía biológica por Crossover (cruce) sobre dos padres (vectores binarios). En esencia es un proceso en el cual un conjunto del espacio de soluciones *E*, de *n* configuraciones, que son llamados "padres", se manipulan para crear un conjunto de descendientes de *m* nuevas configuraciones. La creación de la descendencia involucra extraer y combinar características de las configuraciones padres.

La Recombinación tiene algunas propiedades que es bueno mencionar [11]:

- Respeto: los descendientes heredan todas las características básicas comunes entre todos los padres.
- Variedad: los descendientes toman características compatibles de los padres.
- Transmisión: cada una de las características que exhiben los descendientes, se encuentran en al menos un padre.

2.3.3.5.- Evaluación

El proceso de evaluación consiste en dar a cada configuración (Agente) un peso de importancia con respecto a los demás. Fundamentalmente se tiene en cuenta el valor de la configuración dado por la función objetivo [3].

2.3.4.- Algunas Diferencias Entre Los AG y los AM

Dentro de las diferencias que podemos encontrar entre un AM y un AG, debemos mencionar primeramente la base teórica. Mientras los AG se basan en la teoría evolutiva de Darwin, la que sostiene la selección natural como proceso de mejoramiento de la población, los AM se basan en la teoría de Lamarck de transmisión cultural, es decir, que los individuos son capaces de mejorar de una manera complementaria a la genética, pudiendo aprender de sus vecinos.

La inclusión de optimizadores locales refleja el aprendizaje de un individuo en el medio, el cual es transmitido genéticamente a sus descendientes.

El operador de Recombinación en un AM contiene información relevante sobre el problema, es decir, son "inteligentes" a la hora de realizar el cruce de información entre padres, aprovechando el conocimiento y logrando resultados más rápidos.

Los AM a diferencia de los AG, pueden o no tener un operador de Mutación en el proceso, puesto que en un AM existen los meta-operadores de mutación (optimizadores locales) dentro del proceso de evolución, los cuales pueden ser vistos como mutaciones iterativas a las configuraciones resultantes de la Recombinación [10].

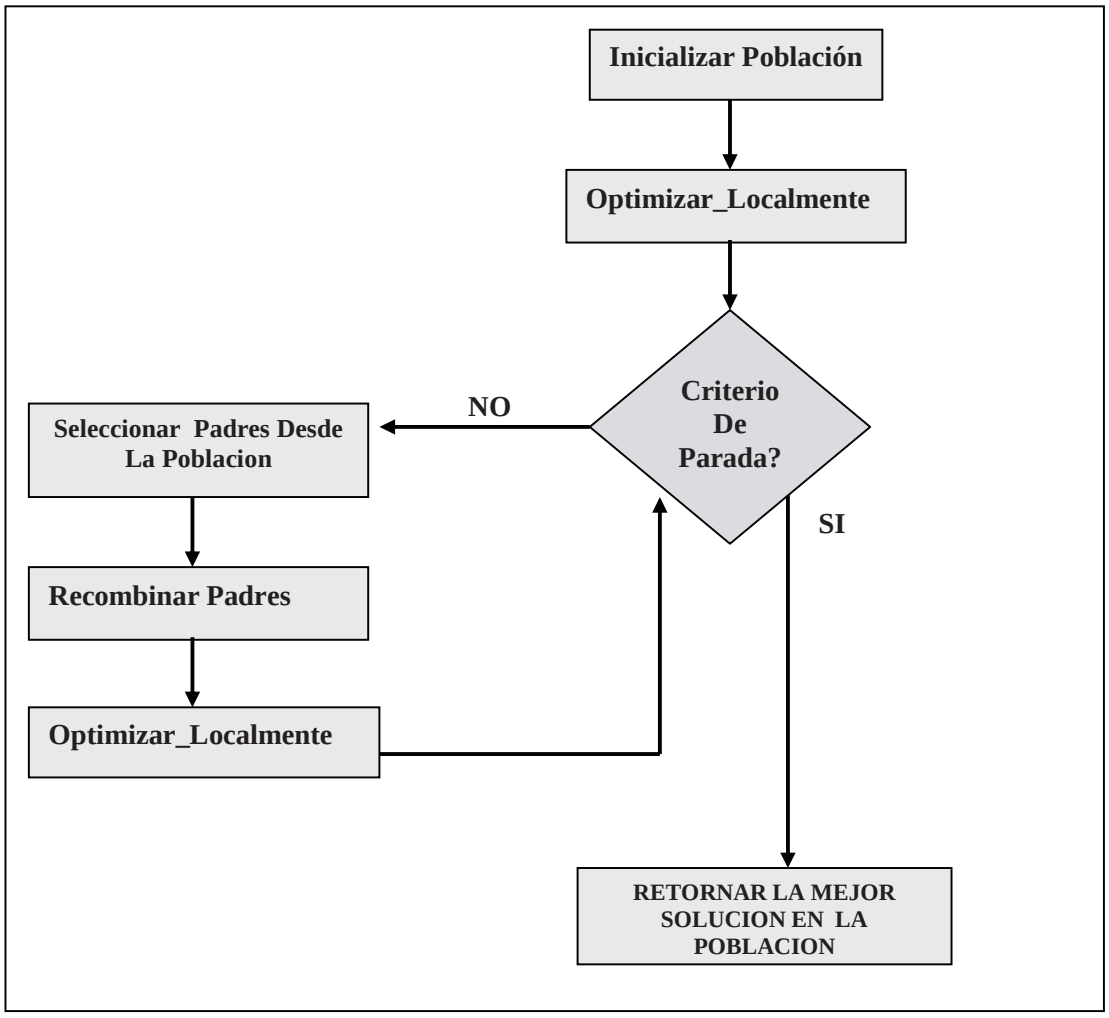


Figura 2.5.- Estructura de un Algoritmo Memético.

CAPITULO 3. DESCRIPCION DEL PROBLEMA

3.1.- LOS FIXTURES DEPORTIVOS Y SUS CARACTERISTICAS

El problema del armado de Fixtures deportivos es un problema cercano a la gente, por ende entendible por la mayoría debido a la relación que existe entre las personas y las ligas o torneos de algún deporte masivo, como el Fútbol, Baloncesto, Tenis, etc. [12].

Un Torneo Deportivo está formado por entidades llamados equipos, cuya función es participar enfrentándose en partidos en una fecha determinada llamada ronda. Existen distintos tipos de torneos, siendo uno de los más populares el denominado *Round Robin,* donde todo equipo *x* juega contra todos los demás la cantidad de veces que sea definida. Si la cantidad de veces es uno, se trata del llamado *Single Round Robin (SRR),* si la cantidad de veces es dos, se denomina *Double Round Robin o DRR (partido y revancha).*

Partimos de la base que un equipo define un lugar en donde juega de local (generalmente es el lugar de procedencia del equipo), los partidos deberán jugarse en uno de los estadio de los contendientes, pudiendo intercambiar la localía en caso del DRR [13,21].

Ejemplo:

Para un Single Round Robin con 8 equipos, se necesitarán (8-1) rondas para enfrentar a todos contra todos.

3.1.1.- Restricciones Generales Para Un Armado De Fixtures

Existe una gran diversidad de requisitos para este problema. Las razones pueden ser muchas y muy importantes, como aumentar el atractivo de la competición y ayudar a los equipos económicamente.

A continuación se describirán los requisitos más comunes para el armado de Fixtures [21]:

- Para hacer un campeonato justo, los equipos deberán jugar la misma cantidad de partidos de visita que de local.
- Dos equipos *x* e *y* no pueden jugar de local en la misma ronda. Esto puede deberse a que comparten estadio. Esta restricción es casi innecesaria puesto que una fecha puede jugarse en tres días diferentes consecutivos.
- Un Equipo *x* debe jugar por obligación de local o visitante en una ronda determinada. Esto se puede dar cuando un estadio está reservado para algún evento, o se encuentra en reparaciones.
- Dos equipos *x* e *y* no pueden enfrentarse en determinadas rondas. Esto se hace para lograr que los partidos "clásicos", o bien los más atractivos popularmente hablando se jueguen en rondas decisivas del campeonato. Así acudirá un mayor número de espectadores a los partidos y el campeonato tome importancia.
- Dos equipos *x* e *y* deben enfrentarse en determinada ronda. Es similar a la anterior y se hace generalmente para retrasar el enfrentamiento de los equipos que en general tienen un mejor rendimiento en los Torneos, con el fin de decidir lo más tarde posible a un campeón y mantener así la expectativa de los hinchas.
- En el caso que se juegue un Double Round Robin, se puede pedir que el Fixture sea **Espejado (MDRR)**, es decir, que la segunda vuelta sea igual que la primera invirtiendo las localías de los equipos. También es posible que no se requiera esto, sino que, después del enfrentamiento entre *x* e *y* pasen a lo menos una cierta cantidad de partidos antes de la revancha. Generalmente se hace en los campeonatos internacionales.
- Minimizar las distancias recorridas por los equipos para completar sus partidos. Con el objetivo de reducir los costos de viaje y usar el tiempo en que se deja de viajar para el descanso de los jugadores o para entrenamientos preparatorios de los partidos siguientes.

3.1.2.- Espacio De Búsqueda

El espacio de búsqueda representa el conjunto de las posibles soluciones a un problema determinado. Para el problema de armado de Fixtures no existe una formula general que nos diga cuantos Fixtures SRR son posibles para *n* equipos. La *Tabla 3.1* muestra la cantidad de Fixtures posibles para torneos SRR y diferentes valores de *n* [22].

Nº de Equipos	Soluciones Posibles
6	720
8	40320
10	3628800
12	479001600
14	8.7178291×10^{10}
16	2.0922789×10^{13}
18	6.4023737×10^{15}
20	2.432909×10^{18}

Tabla 3.1.- Espacio de Soluciones para un SRR

Podemos ver que el espacio de soluciones crece exponencialmente con el aumento de los equipos, por lo que se justifica el uso de una Metaheurística como la de Algoritmos Meméticos.

3.1.3.- Representaciones al Problema

La representación del problema es un paso sumamente importante en la resolución, puesto que nos da un lenguaje con reglas a seguir para comunicarse con la técnica de optimización, nos ayuda a visualizar de mejor forma el problema y los procedimientos a seguir para resolverlo. Lo más importante es que nos permite crear un modelo para el algoritmo de optimización.

En la literatura existen varias representaciones para este problema, a continuación se entregará un resumen de ellas para el caso de un torneo Single Round Robin [15].

3.1.3.1.- Tablas o Matrices

La representación más recurrente es la matricial. Se puede ver fácilmente un Fixture como una tabla de dimensiones *(n-1) * n,* en donde cada fila representa una ronda y cada columna un equipo. Esta matriz se llama M*r,e,* y una celda contiene un partido de la ronda *r,* en donde un equipo juega contra el equipo *e.*

Para una mejor lectura, los equipos se designarán como A, B, C, D y las rondas del 1 al 3. El signo "@" significa que el equipo que sigue al signo hace de local, es decir, "@A" significa "A es local".

Ronda/Equipos	A	B	C	D
1	B	@A	D	@C
2	C	D	@A	@B
3	@D	C	@B	A

Figura 3.1.- Representación de un Fixture con una Matriz.

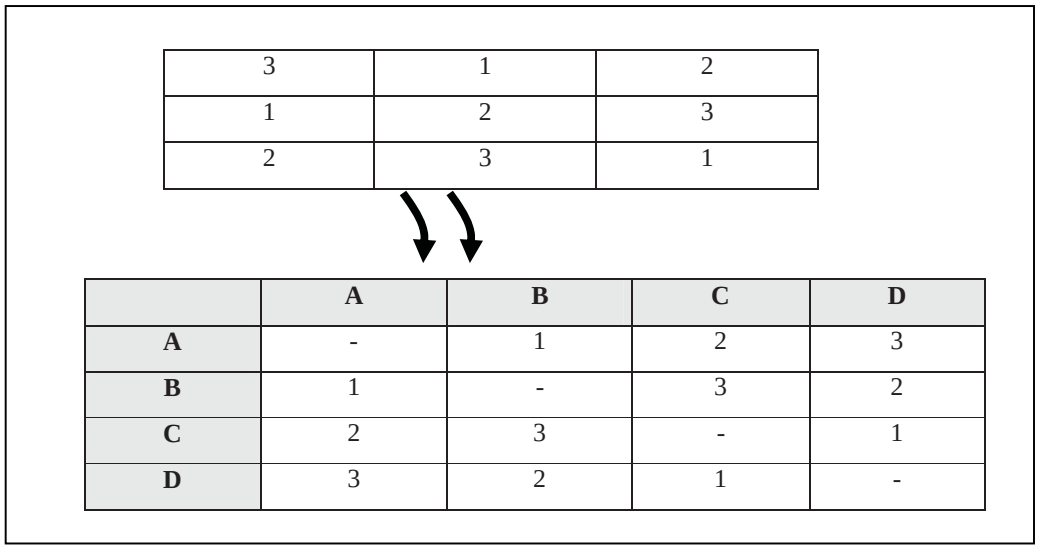
La *Figura 3.1* nos muestra un SRR de cuatro equipos con 3 rondas. Esto permitirá que los equipos jueguen todos contra todos una sola vez. Para un MDRR, sólo se deben cambiar las localías manteniendo el orden de los rivales para cada equipo.

3.1.3.2.- Cuadrados Latinos

Un Cuadrado Latino es una matriz de *n*n*, donde en cada posición se puede colocar cualquier valor entero entre *1* y *(n-1)* con la condición de que ninguno aparezca más de una vez en la misma fila o columna. Siguiendo algunos pasos podemos obtener un Fixture SRR:

- Construir un cuadrado latino de *(n-1)*(n-1)* en el cual no se repita ningún número en la diagonal principal.
- Construir una nueva fila y una nueva columna copiando los valores de la diagonal.
- Dejar en blanco los valores de la diagonal.

Ejemplo: En la *Figura 3.2* se muestra la construcción de un Fixture.



3	1	2
1	2	3
2	3	1

	A	**B**	**C**	**D**
A	-	1	2	3
B	1	-	3	2
C	2	3	-	1
D	3	2	1	-

Figura 3.2.- Fixture en un cuadrado latino.

El resultado de seguir los pasos fue un Fixture para cuatro equipos, donde los números en las celdas representan las rondas en que los equipos deben enfrentarse. Por ejemplo, en la matriz resultado, la celda {3,2} enfrenta a los equipos C y B, el número tres que se encuentra en dicha celda nos dice que estos equipos se enfrentarán en la tercera ronda.

3.1.3.3.- Grafos

Los grafos son otra forma de representación de Fixtures. Para un torneo de *n* equipos se explicará como colorear arcos para hacer la correspondencia con los partidos [21].

Coloreo de Arcos

Un grafo completo de *n* nodos (por los *n* equipos) G = (V,E) y un coloreo de arcos que representan los partidos según la ronda, conforman un Fixture. Para un *n* par, es claro que se necesitan *n-1* colores para diferenciar las *n-1* rondas.

Para armar los encuentros o partidos de la ronda *i*, se toman los arcos pintados con el color *i* (los colores se definen con un número correspondiente a la ronda). Cada uno de los arcos por separado representa un partido.

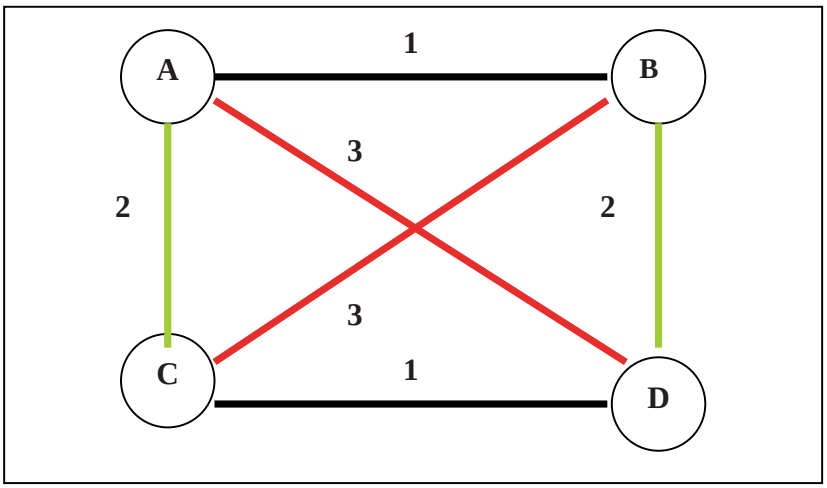


Figura 3.3. Grafo de un Fixture.

De la *Figura 3.3* podemos deducir que:

- A, B, C y D son los equipos participantes.
- Los arcos (líneas que unen los nodos) representan los partidos.
- Los números asociados a los arcos representan la ronda en que se deben enfrentar, y se deben colorear de diferente forma.

3.2.- TRAVELING TOURNAMENT PROBLEM

M. Trick, G. Nemhauser, y K. Easton presentaron una clase de problema llamado *Traveling Tournament Problem (TTP*), el cual consistía en el estudio del armado de Fixtures deportivos teniendo en consideración ciertas restricciones y el objetivo de minimizar las distancias recorridas [14].

El TTP se basa en las distancias que tienen que recorrer los equipos de la MLB de Baseball (Mayor League Baseball) de Estados Unidos. La MLB está dividida en dos partes, una de ellas, la National League (NL), posee 16 equipos ubicados en ambas costas de Estados Unidos y Canadá. Las instancias del NL para los cuales se conocen resultados, se generaron tomando subconjuntos de estos 16 equipos. De esta manera la instancia NL*x* está formada por las distancias entre los primeros *x* equipos. Hasta el momento sólo se conoce el valor óptimo para NL4, NL6 y NL8 [15].

El Mirrored Traveling Tournament Problem (MTTP) es un caso especial de TTP, al cual se le agrega una característica común en casi todos los torneos de Latino América, y es que el Fixture sea un DRR espejado o Mirrored Double Round Robin (MDDR). Trabajos en Brasil han tratado este problema con buenos resultados [21].

3.2.1.- Definición Del Problema

En la S*ección 3.1* se definieron los torneos SRR y DRR, en donde los *n* equipos deben enfrentarse todos contra todos. El Fixture se divide en *(n-1)* rondas para el SRR y 2*(n-1)* rondas para el DRR, donde cada ronda contiene *n*/2 partidos. En cada partido un equipo debe jugar de local (Home) y otro de visitante (Away). Se define HAP (Home Hawai Pattern), como la secuencia de estados que tomará un equipo durante un torneo. Por ejemplo, un HAP igual a **LVVL** significa que el primer partido será de local, seguido por dos partidos de visita y uno de local.

Se define D, como una matriz de (*n* x *n)* que contiene las distancias entre los equipos involucrados en el torneo, tal que la distancia entre A y B es igual a la distancia entre B y A (Simetría). Los viajes de los equipos pueden pertenecer a unos de los siguientes tipos [6]:

- En la ronda *r* el equipo A juega de local, y en la ronda *(r + 1)* juega de visitante en B. La distancia que se tomará de la matriz será la que está entre A y B.
- En la ronda *r* el equipo A juega de visita en B, y en la ronda *(r + 1)* juega de visita en C. La distancia que se tomará de la matriz será la que está entre B y C. Esta modalidad recibe el nombre de ***road trip,*** y su longitud será la cantidad de partidos de visita consecutivos, no las distancias recorridas.
- En la ronda *r* el equipo A juega de local, y en la ronda *(r + 1)* juega de local. La distancia que se tomará será cero. A esta modalidad se le llama ***home stand,*** y su longitud es la cantidad de partidos de local consecutivos, y no a las distancias recorridas.

Cada equipo comienza y termina el torneo en su casa, lo que significa que se debe agregar la ronda 0 y la 2*(*n-1).* La justificación de esto es que si en el primer partido el equipo A juega de visita en B, debe contabilizarse el desplazamiento del equipo A hasta B. Del mismo modo, si en el último partido del torneo el equipo A juega de visitante, debe contabilizarse la distancia recorrida por el equipo para llegar a casa [14].

3.2.2- Condiciones Del Problema

Para el desarrollo del problema se deben tener en cuenta las siguientes entradas y restricciones generales [14]:

- *n* = número de equipos.
- D = matriz de distancias de *n* x *n*.
- U = parámetro entero.

El parámetro U nos sirve para definir un valor fijo, que representará la cantidad de partidos consecutivos que un equipo cualquiera puede jugar en una determinada condición dentro del torneo (Local o Visita). Por ejemplo, si el parámetro U = 1, significa que los equipos jugarán

alternadamente de local y visita. El MTTP tiene sentido para valores de U >1, ya que se puede ahorrar kilómetros si unimos dos o más partidos de visita en un viaje del equipo. El presente trabajo asumirá un valor de U = 3.

Existe un caso especial, cuando U = *n* el problema se parece al TSP, por que en este caso el equipo podría jugar todos sus partidos de visita y luego regresar a su estadio. Este valor de U es teórico, ya que en la realidad no es factible de realizar [13,14].

Las restricciones del problema son las siguientes:

- Minimizar las distancias recorridas por los equipos en el torneo.
- Ningún equipo puede jugar más de U partidos seguidos de local o visita.
- El Fixture debe ser **espejado,** o que la segunda parte del torneo se juegue en el mismo orden pero cambiando localías.

La salida del problema debe ser un Fixture MDRR, que cumpla con las restricciones antes mencionadas y que minimice las distancias recorridas por los equipos. Además se pueden agregar restricciones como [14]:

- Cualquiera de las expuestas en la *Sección 3.1.1.*
- El presente proyecto no incorpora esas restricciones.

3.2.3.- Formulación Matemática

Para formular el problema matemáticamente se deben recordar algunos aspectos mencionados anteriormente. El primero es que se trabajará con una matriz como estructura de datos que contendrá el Fixture, tal como se explicó en la *Sección 3.1.3.1.* Y el segundo es que las restricciones se expresarán como en el CSP, agregando las restricciones del MTTP a las ya expuestas que son propias de un MDRR.

A continuación se modela el MTTP como un CSP, definiendo la Tupla (X,D,C) [7]:

- X contiene las variables $M_{r,e}$ (e = 1...*n*, *r* = 1..R) R = *n* – 1
- D*re={-n,...,-1,1,....,n}\ {e,-e}* con *e*=1..n, *r*=1...n-1)
- C contiene las siguientes restricciones :

Restricciones del MDRR

- $Mr,e \neq Ms,e \ \forall\ r \neq s$ con e = 1..*n*; r , s = 1...*n*-1. (3.1)
- $[\ i \neq j \Rightarrow |Mr,i| \neq |Mr,j|\ ]$ $\forall\ r :1 \leq r \leq n\text{-}1,\ (i,j= 1..n)$ (3.2)
- $[\ Mr,i = j \Leftrightarrow Mr,j=-i\ ]$ $\forall\ r :1 \leq r \leq n\text{-}1,\ (i,j= 1..n)$ (3.3)

Agregando las restricciones del MTTP, fijando U = 3.

- $|Mr,e| \neq |M^{r+n}_{r+1},e|$ (3.4)
- $[M_{r,e} > 0 \wedge M_{r+3,e} > 0] \Rightarrow [M_{r+1,e} < 0 \vee M_{r+2,e} < 0]$ (3.5)
- $[M_{r,e} < 0 \wedge M_{r+3,e} < 0] \Rightarrow [M_{r+1,e} > 0 \vee M_{r+2,e} > 0]$ (3.6)

Número	Restricción
3.1	Todo equipo juega dos veces contra todos los demás
3.2	En cada ronda todos juegan con un equipo distinto
3.3	Si A juega de local contra B, entonces B juega de visitante contra A
3.4	Ningún par de equipos juega entre sí nuevamente antes de *n* rondas. Fixture espejado.
3.5	Ningún equipo juega más de 3 partidos seguidos de local
3.6	Ningún equipo juega más de 3 partidos seguidos de visitante.

Tabla 3.2.- Restricciones del TTP

CAPITULO 4. DISEÑO DE LA SOLUCION

Para diseñar un Algoritmo Memético se deben considerar algunos factores. Podemos decir que un AM, más que una técnica que deba aplicarse directamente como dice una formula, es una estrategia de búsqueda que usa métodos subordinados para realizar el proceso de optimización. Por esta razón los AM pueden variar de una implementación a otra y es un campo de constantes novedades. Los AM dependen de la bondad de sus partes, pudiendo estas ser reemplazadas a medida que aparezcan técnicas superiores, sin dejar de aplicar la estrategia Memética.

Independientemente de cuán sofisticado pueda ser el diseño de un AM, existen algunos componentes que deben ser incluidos [3]:

- Una forma de representar las configuraciones (Agentes) del problema.
- Una manera de crear las configuraciones de la población inicial.
- Una función de evaluación, que permita ordenar los individuos de acuerdo a su valor asociado de la función objetivo.
- Valores de los siguientes parámetros:
    - Tamaño de la Población: cantidad de configuraciones que se generarán.
    - Cantidad de configuraciones que se utilizarán para la recombinación.
    - Cantidad de configuraciones que se reemplazarán en una generación.
    - Estrategia de Selección : forma de seleccionar los padres de una generación entre todos los agentes que componen la población.
- Estrategia de Búsqueda Local, que permita imitar la evolución memética.
    - Movimientos: operaciones sobre las configuraciones que definen vecindades.
- Operador de Recombinación, que mezcla las características de los padres para crear otras configuraciones (hijos).
- Criterio de parada, que determina cuando el algoritmo deja de iterar.

A continuación se describirán los componentes que se utilizarán en el Algoritmo Memético, para resolver el *Mirrored Traveling Tournament Problem.*

4.1.- REPRESENTACION DE LAS SOLUCIONES

Para representar las soluciones del problema, es decir, los Fixtures MDRR que serán los agentes de nuestra población se utilizará una matriz de *n* x 2*(n-1).* En lenguaje biológico esta matriz se denomina "*Cromosoma*", y las celdas que la forman se denominan "*Genes*".

Las filas de esta Matriz (M*r*,*e*) representan las rondas del Fixture, y las columnas contienen a los *n* equipos.

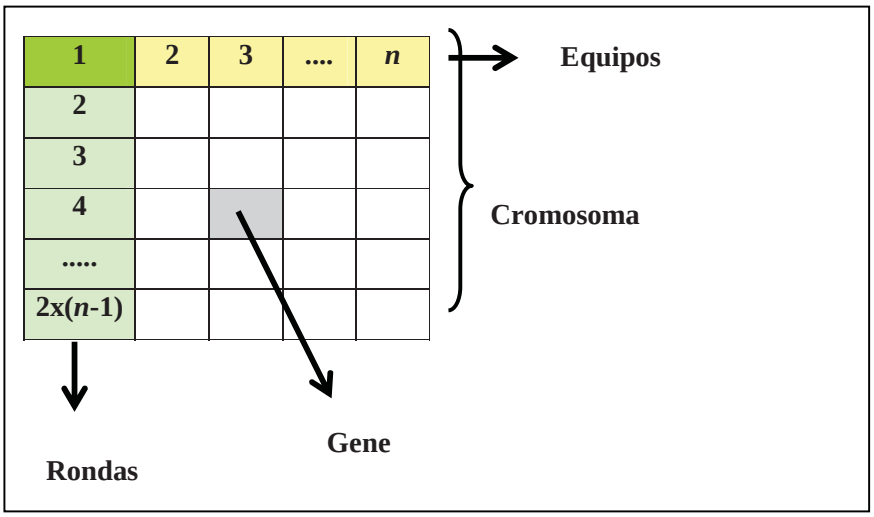


Figura 4.1.- Representación de las Soluciones

4.2.- FUNCION OBJETIVO

El primer punto que se debe considerar a la hora de resolver un problema, es la razón fundamental por la que se está diseñando un algoritmo. En este caso lo que se persigue es minimizar una función que mide los kilómetros recorridos por los equipos, durante el desarrollo de un MDRR. Además dicha función nos servirá como criterio de comparación entre dos o más soluciones.

Siguiendo los tipos de viajes planteados en la *Sección 3.2.1*, se define la función objetivo *F* como la suma de todos los recorridos de los *n* equipos involucrados en el torneo.

Recordando lo definido en la *Sección 3.2.1, D* representa la matriz de distancias tal que $D_{i,j,}$ muestra la distancia que existe entre el estadio del equipo *i* y el estadio del equipo *j*.

Por otro lado el MTTP indica que todos los equipos inician y terminan el torneo en casa. En otras palabras, se agrega la ronda *0* y *2x(n-1)* donde todos los equipos son locales. Luego se define $T_{k,r}$ como la distancia que debe recorrer el equipo *k* desde el estadio donde juega en la ronda *(r – 1)* al estadio donde juega en la ronda *r*, de la siguiente manera [15].

$$\mathbf{T}k,r = \begin{cases} DIST_{|Mr-1,k|,|Mr,k|} & \text{si } (Mr\text{-}1,k < 0) \wedge (Mr,k < 0) \quad (i) \\ DIST_{|Mr-1,k|,k} & \text{si } (Mr\text{-}1,k < 0) \wedge (Mr,k > 0) \quad (ii) \\ DIST_{k,|Mr,k|} & \text{si } (Mr\text{-}1,k > 0) \wedge (Mr,k < 0) \quad (iii) \\ 0 & \text{si } (Mr\text{-}1,k > 0) \wedge (Mr,k > 0) \quad (iv) \end{cases} \tag{4.1}$$

(i) El equipo *k* juega de visitante en la ronda *r* y *r-1*. Entonces la distancia buscada es la que separa a los rivales de *k* en *r* y *r-1*.

(ii) El equipo *k* juega de local en la ronda *r* y de visitante en *r-1*. Luego, la distancia buscada es la que separa al estadio de *k* con el estadio de su rival en la ronda *r-1*.

(iii) El equipo *k* juega de local en la ronda *r-1* y de visitante en *r*. Luego, la distancia buscada es la que separa al estadio de *k* con el estadio de su rival en la ronda *r*.

(iv) El equipo k juega de local en las rondas *r* y *r-1*. Por lo tanto, permanece en su estadio y la distancia es cero.

Finalmente definimos la función de objetivo *F*, de la siguiente manera [15]:

$$F(\mathrm{x}) = \sum_{e=1}^{n} Ce \tag{4.2}$$

Donde Ce indica la distancia total recorrida por el equipo *e* a lo largo de todo el torneo. El valor de los Ce (e=1...n) se calcula de esta forma:

$$Ce = \sum_{r=1}^{n} Te, r \tag{4.3}$$

4.3- GENERACION DE LA POBLACION INICIAL

Según la *Figura 2.5*, el primer paso del AM es el proceso de inicialización de la población de Agentes. Como se explicó en la *Sección 2.3.3.1*, esta inicialización consiste en la creación de las configuraciones (individuos) y su posterior optimización local (Agentes). Este proceso se divide en 4 etapas:

1.- Se deben determinar los partidos que se jugarán por ronda mediante el Método del Polígono.

2.- Se guardarán los partidos obtenidos en la Matriz $M_{r,e}$.

3.- Se asignarán las localías aleatóriamente o siguiendo algún patrón que respete las restricciones del MTTP.

4.- Se aplicará un optimizador local a las configuraciones para crear Agentes.

4.3.1.- Método Del Polígono

El método del polígono permite crear Fixtures de manera aleatoria. Es un método bastante fácil y eficiente, es decir, las configuraciones generadas no necesitan reparación [21].

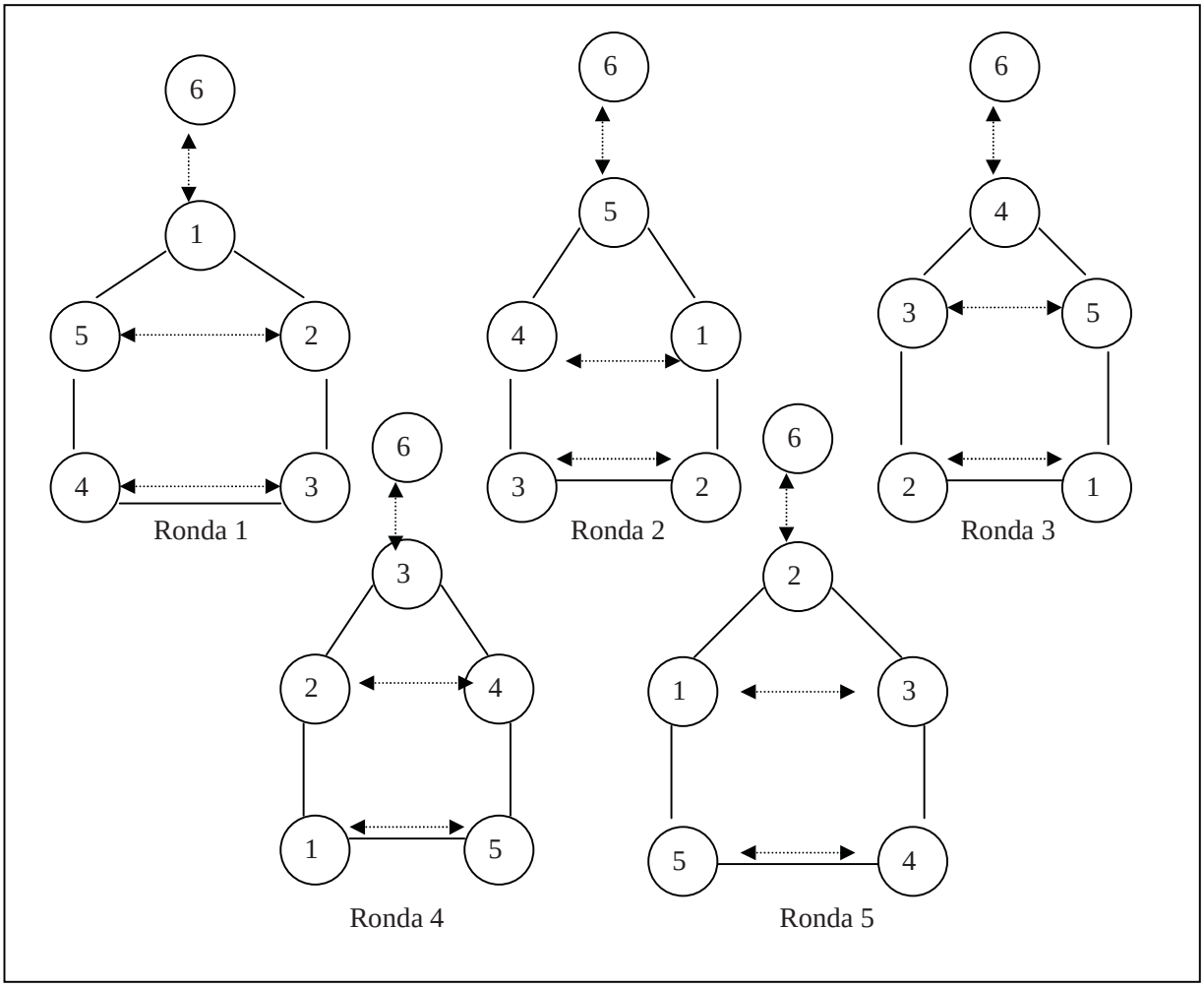


Figura 4.2.- Ejemplo de Método del polígono para construir Fixtures.

En la *Figura 4.2* se muestra la primera de tres fases en la construcción de un Fixture deportivo. Las flechas indican los partidos que se jugarán en cada ronda, por ejemplo, en la Ronda 1 se enfrentará el equipo 6 contra el equipo 1, el equipo 2 contra el 5 y el equipo 3 contra el 4.

4.3.2.- Asignación De Partidos A La Estructura De Datos

La segunda fase define la estructura en la que se guardarán los partidos, en este caso la Matriz $M_{r,e}$ (*Tabla 4.1*) [21].

Ronda/Equipos	1	2	3	4	5	6
1/6	*6*	*5*	*4*	*3*	*2*	*1*
2/7	*4*	*3*	*2*	*1*	*6*	*5*
3/8	*2*	*1*	*5*	*6*	*3*	*4*
4/9	*5*	*4*	*6*	*2*	*1*	*3*
5/10	*3*	*6*	*1*	*5*	*4*	*2*

Tabla 4.1.- Matriz M*r,e*

La estructura traduce lo obtenido por el método del polígono, donde cada fila representa una ronda y las columnas muestran el orden de los rivales para cada equipo.

4.3.3.- Asignar Localías

Se asignarán las localías de manera aleatoria o siguiendo algún patrón que permita la factibilidad de la solución. El siguiente es un método que trata de agrupar el máximo de partidos posibles en un *road trip*, así es posible disminuir la distancia recorrida por los equipos [21].

La primera ronda se asigna de manera aleatoria. Ahora se asignará las localías comparando a los equipos *e1* y *e2*, registrando la cantidad de partidos consecutivos que tiene cada uno en cierta condición (local o visitante). Se define *ni* como tal cantidad para un equipo *i*. La comparación incluye los siguientes casos:

Caso 1: *ne2 >ne1*

- Si *e2* jugó su último partido de local, se programa el partido en el estadio de *e1*.
- En otro caso, se programa en el estadio de *e2*.

Caso 2: *ne2 < ne1*

- Si *e1* jugó su último partido de local, el partido se programará en el estadio de *e2*.
- En otro caso el partido se programará en el estadio de *e1*.

Caso 3: *ne2 = ne1*

- Si *e1* ha jugado su último partido de local y *e2* de visitante, el partido se programará en el estadio de *e2*.
- Si *e2* ha jugado su último partido de local y *e1* de visitante, el partido se programará en el estadio de *e1*.
- En otro caso, se asigna aleatóriamente el estadio del partido.

Ejemplo: Ronda 1 se asigna aleatóriamente.

Ronda 2

e1 vs e4 ⇒ ne1 = ne4, misma condición del último partido, se asigna aleatóriamente = @e4

e2 vs e3 ⇒ ne2 = ne3, misma condición del último partido, se asigna aleatóriamente = @e2

e5 vs e6 ⇒ ne5 = ne6, e5 fue local en su último partido, e6 fue visitante = @e6

Ronda 3

e1 vs e2 ⇒ ne1 = ne2, e2 fue local en su último partido, e1 fue visitante = @e1

e3 vs e5 ⇒ ne3 < ne5, e5 fue visitante en su último partido = @e5

e4 vs e6 ⇒ ne4 < ne6, e6 fue local en su último partido = @e4

Ronda 4

e1 vs e5 ⇒ ne1 = ne5, misma condición en el último partido, se asigna aleatóriamente= @e5

e2 vs e4 ⇒ ne2 < ne4, e4 fue visitante en su último partido = @e4

e3 vs e6 ⇒ ne3 = ne6, misma condición en el último partido, se asigna aleatóriamente = @e3

Ronda 5

e1 vs e3 ⇒ ne1= ne3, e3 fue local y e1 visitante = @e1

e2 vs e6 ⇒ ne2 = ne6, misma condición en el último partido, se asigna aleatóriamente = @2

e4 vs e5 ⇒ ne4 > ne5, e4 fue local = @e5

Ronda/Equipos	1	2	3	4	5	6
1/6	*@6*	*5*	*4*	*@3*	*@2*	*1*
2/7	*@4*	*3*	*@2*	*1*	*@6*	*5*
3/8	*2*	*@1*	*@5*	*6*	*3*	*@4*
4/9	*@5*	*@4*	*6*	*2*	*1*	*@3*
5/10	*@3*	*6*	*1*	*@5*	*4*	*@2*

Tabla 4.2.- Tercera parte en la generación de un Fixture.

4.4.- OPTIMIZADOR LOCAL

La estrategia de búsqueda local es la que representa el aprendizaje de los individuos para convertirse en agentes. Esto lo hace a través de la exploración de vecindarios de las soluciones generadas inicialmente, o los hijos que salgan de la recombinación de 2 o más soluciones padres. También el optimizador local puede verse como un meta-operador de mutación, cambiando aleatoriamente la configuración de la solución de manera iterativa o como lo determine la Heurística [10].

4.4.1.- Estrategia Del Operador De Búsqueda Local

En la *Figura 2.5* se muestran los componentes de un AM, donde se ven dos fases de optimización local. La primera es cuando se generan las soluciones iniciales, y la segunda después de aplicar la Recombinación.

El método del polígono nos entrega soluciones que respetan las restricciones del TTP, por lo tanto, la infactibilidad de una solución inicial está ligada claramente a la asignación de las localías. Aunque para el optimizador local no es necesariamente un problema [11].

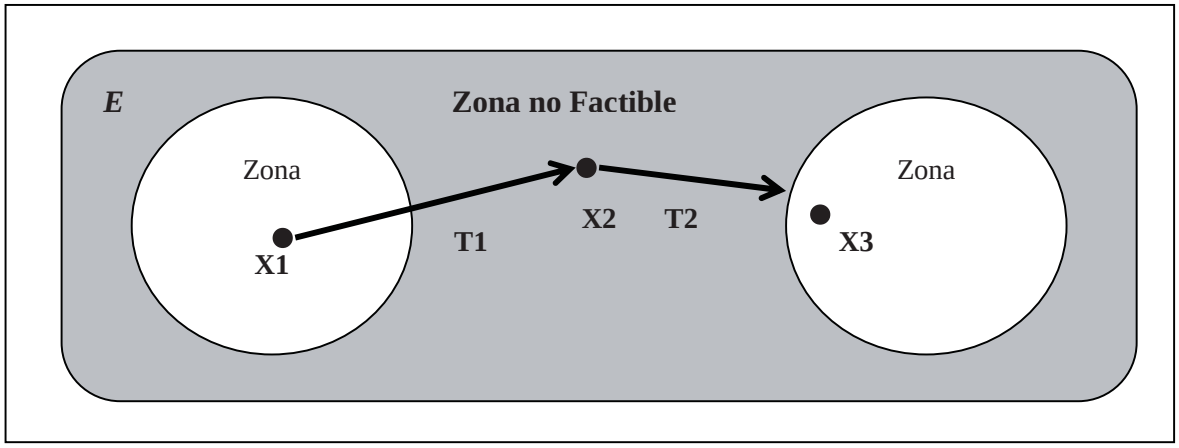


Figura 4.3.- Espacio Total de Soluciones

La *Figura 4.3* muestra el espacio total de soluciones *E*, haciendo una separación entre el espacio de soluciones factibles y no factibles. Los puntos *X1, X2* y *X3* representan soluciones pertenecientes a un sub-espacio. *T1* y *T2* son transformaciones, la primera lleva de *X1* a *X2* y la segunda lleva de *X2* a *X3.*

El optimizador local aplica transformaciones a las soluciones iniciales, realizando un viaje a través del espacio de soluciones. Es posible que una transformación nos lleve a una solución no factible, pero en vez de desechar la solución, se puede seguir ese camino hasta llegar a una región factible mediante transformaciones sucesivas. Esta estrategia nos permitirá ganar en diversidad para que la población no converja a un óptimo local [15].

Cada transformación nos lleva a una vecindad, por lo tanto, distintas transformaciones distribuyen de manera diferente las vecindades y pueden acercar soluciones que con otra transformación estaban muy lejos, o alejar soluciones que estaban cerca. Así la elección de la transformación puede hacer la diferencia entre tener o no tener éxito en la búsqueda y que tan eficiente es el algoritmo propuesto.

A continuación se explicarán diferentes transformaciones para el MTTP, cada una tiene su nivel de eficiencia según lo anteriormente dicho.

4.4.2.- Vecindarios

Los vecindarios dependen de las transformaciones aplicadas como se dijo anteriormente. Por esta razón la existencia de varias transformaciones nos permitirá diversificar la población ya que se explorará en diferentes direcciones del espacio. Para el TTP existen 3 transformaciones, de las cuales la que ha entregado mejores resultados es la que da origen al llamado vecindario principal [15].

4.4.2.1.- Intercambio Parcial De Rondas – Vecindario Principal

El Intercambio Parcial de Ronda (IPR), define una vecindad IPR(x) de un Fixture $x \in E$ y consiste en todos los Fixtures que se pueden obtener aplicando a x un movimiento IPR. Un movimiento IPR es una permutación de cuatro partidos entre dos rondas (*r1, r2*) que involucra a cuatro equipos (*e1, e2, e3, e4*). Para que se pueda aplicar un movimiento IPR a un Fixture, este debe satisfacer las siguientes propiedades [15,21]:

- x.rival(*e1,r1*) = x.rival(*e2,r2*) = *e3*
- x.rival(*e1,r2*) = x.rival(*e2,r1*) = *e4*

Donde x.rival(e,r) indica el equipo contra el que juega e en la ronda r en el Fixture x, es decir, $|M^{x}_{r,e}|$.

Como ejemplo utilizaremos un Fixture arbitrario con valores en las celdas relevantes. Se aplicará IPR cumpliendo las condiciones antes mencionadas.

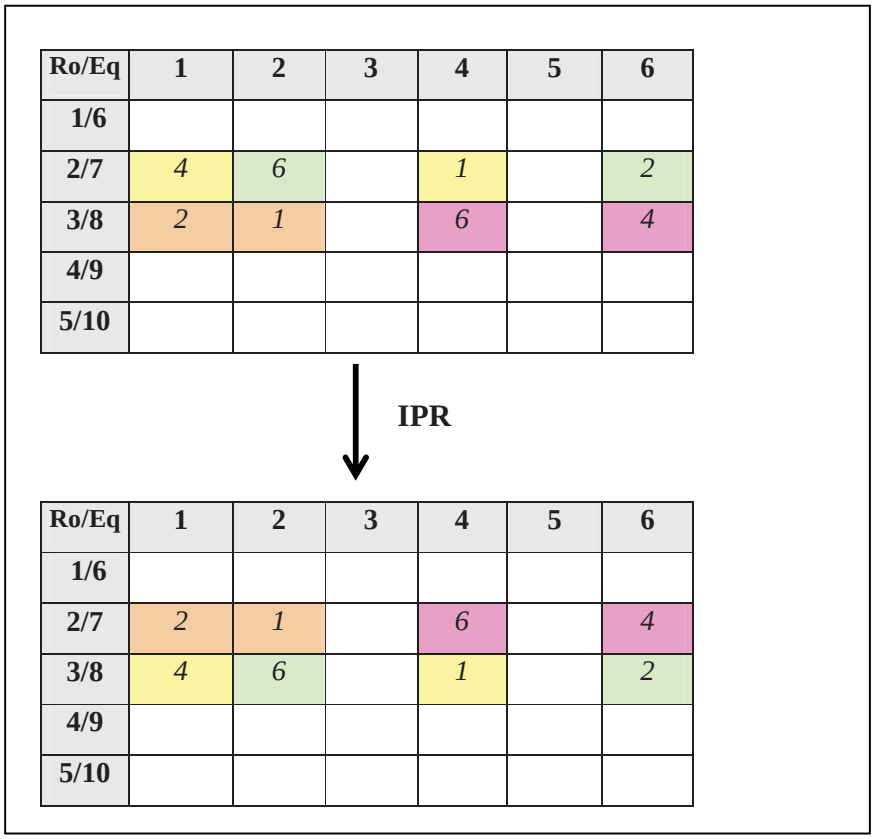


Figura 4.4.- Aplicación de IPR

En la *Figura 4.4* se muestra la generación de un vecino de un Fixture *X*, mediante la aplicación del Intercambio Parcial de Ronda. Las condiciones se cumplen tomando *e1 =1, e2 = 6, e3 = 2 y e4 = 1*, para las rondas *2 y 3 (r1 y r2).* Luego se puede aplicar el intercambio, lo que significa que los partidos que se jugaban en la ronda 3, pasan a jugarse en la ronda 2 sin afectar la estructura de los demás partidos. Así se logra un Fixture *Y* que cumple con:

- y.rival($e1$,$r1$) = y.rival($e2$,$r2$) = $e4$ = x.rival($e1$,$r2$) = $|M^{x}_{r2,e1}|$
- y.rival($e1$,$r2$) = y.rival($e2$,$r1$) = $e3$ = x.rival($e1$,$r1$) = $|M^{x}_{r1,e1}|$
- El resto de los partidos se mantienen igual.

Se puede observar que al aplicar IPR sobre un Fixture, este no pierde las propiedades de un MDRR, aunque si puede ocurrir que la solución generada no sea válido para el MTTP. En particular se puede obtener un vecino que rompa cualquiera de las restricciones definidas por *3.4, 3.5, 3.6.*

Existe una variante de IPR que consiste en que además de cambiar los partidos de rondas, también está la posibilidad de cambiar las localías de los partidos. Esto afecta al MDRR ya que se cambian las localías de la segunda ronda. Cuando se ve que se puede aplicar IPR a un Fixture, se evalúan todas las combinaciones de localías que se pueden dar para los cuatro equipos (2^4), seleccionando aquella con la que se obtiene mejor resultado [15].

El IPR no siempre es posible, la probabilidad de poder usarlo es muy poca para torneos de pocos equipos, formalmente se aconseja usarlo para $n > 8$. Si no se puede usar, se utilizarán otras transformaciones [21].

4.4.2.2.- Intercambio De Rondas

La vecindad Intercambio de Rondas IR(x) de un Fixture $x \in E$, consiste en todos los Fixtures que se puedan obtener a partir de x permutando un par de rondas *r1* y *r2*, es decir, que simplemente se permutan dos filas de la matriz (M) asociada al Fixture *x*. Se puede ver que al aplicar esta transformación no se viola ninguna restricción relacionada con el MDRR (3*.1, 3.2, 3.3*), pero si las referentes al MTTP (3*.4, 3.5, 3.6*).

Para analizar a todos los vecinos de x se deben seleccionar todos los subconjuntos de dos rondas que se puedan encontrar en él. Luego se evalúan todas las permutaciones calculando la diferencia de costo, aunque puede bastar con elegir el camino que muestre la primera evaluación que logre mayor beneficio, y así agilizar la búsqueda [15,21].

Ejemplo:

Fixture *X*

Ro/Eq	1	2	3	4	5	6
1/6	@6	5	4	@3	@2	1
2/7	@4	3	@2	1	6	@5
3/8	2	@1	5	@6	@3	4
4/9	@5	@4	6	2	1	@3
5/10	3	6	@1	@5	4	@2

IR →

Fixture *Y*

Ro/Eq	1	2	3	4	5	6
1/6	@6	5	4	@3	@2	1
2/7	@5	@4	6	2	1	@3
3/8	2	@1	5	@6	@3	4
4/9	@4	3	@2	1	6	@5
5/10	3	6	@1	@5	4	@2

Figura 4.5.- Intercambio de Rondas IR

4.4.2.3.- Intercambio De Localías

La vecindad Intercambio de Localía IL(*x*) de un Fixture $x \in E$ consiste en todos los Fixtures que se pueden obtener a partir de *x*, cambiando la localía de los dos partidos entre un mismo par de equipos *e1* y *e2*. Es decir, que si en *x*, *e1* jugaba de local contra *e2* en la ronda *r1*, y de visitante en la ronda *r2*, las únicas diferencias entre *x* y el vecino que se obtiene permutando las localías de los equipos *e1* y *e2* son simplemente que en este último, *e1* juega contra *e2* de visitante en la ronda *r1* y de local en la ronda *r2*. Esto implica cambiar el signo de los valores de las celdas M[*r1*,*e1*], M[*r2*,*e1*], M[*r1*,*e2*], M[*r2*,*e2*]. Una permutación de este tipo sólo puede violar las restricciones 3.5 y 3.6. [15].

La *Figura 4.6* muestra un ejemplo de esta operación:

Ro/Eq	1	2	3	4	5	6
1/6	*@6*	*5*	*4*	*@3*	*@2*	*1*
2/7	*@4*	*3*	*@2*	*1*	*6*	*@5*
3/8	*2*	*@1*	*5*	*@6*	*@3*	*4*
4/9	*@5*	*@4*	*6*	*2*	*1*	*@3*
5/10	*3*	*6*	*@1*	*@5*	*4*	*@2*

↓ **IL**

Ro/Eq	1	2	3	4	5	6
1/6	*@6*	*5*	*4*	*@3*	*@2*	*1*
2/7	*4*	*3*	*@2*	*@1*	*6*	*@5*
3/8	*2*	*@1*	*5*	*@6*	*@3*	*4*
4/9	*@5*	*@4*	*6*	*2*	*1*	*@3*
5/10	*3*	*6*	*@1*	*@5*	*4*	*@2*

Figura 4.6.- Intercambio de Localías

Nota : En todos los ejemplos de vecindarios sólo se muestra un SRR. Como el MTTP es para un MDRR, estos cambios también deben verse reflejados en la segunda vuelta de partidos, lo que no afecta la factibilidad de la solución.

4.5.- RECOMBINACION

El operador de Recombinación es el encargado de crear agentes a partir de dos o más agentes llamados padres. En un AM este operador es inteligente, es decir, es propio del problema. En el caso del MTTP es bastante complicado usar cualquier operador conocido de crossover, ya que los hijos generados pueden violar de manera severa las restricciones tanto del MDRR como del TTP.

Se explica a continuación el operador a utilizar, el cual tiene una estrategia de reparación incluida, es decir, luego de recombinar los padres se verifica la completitud de los tours (llamamos tour a la programación de un equipo) generados en los hijos.

Puede pasar que la reparación de un hijo sea muy costosa, en ese caso el hijo será desechado. Siendo éste el único caso en que una solución se deseche. En resumen, una solución se desechará sólo si rompe alguna restricción del MDRR, en caso contrario se conservará para la aplicación del optimizador local.

4.5.1.- Método De Crossover

El método de crossover consiste en la recombinación de dos padres *X* e *Y* que representan Fixtures o Agentes, con la posibilidad de comprobar la completitud de los Tours generados para los equipos. El funcionamiento del método es el siguiente:

a. Se generarán dos hijos. Se recorren las estructuras de los padres y se comparan las celdas ubicadas en las mismas posiciones. Si son iguales, se copia el gene repetido en las estructuras de ambos hijos en la posición encontrada en los padres.

b. Se generarán los hijos *XY* e *YX*. Luego los genes restantes vacíos se rellenarán con valores aleatorios con la dificultad que ello representa. La complicación radica en que para instancias grandes del problema, es baja la probabilidad de lograr completar un Fixture desde uno con genes ya asignados.

c. Si al terminar la recombinación, aun existen genes vacíos en los hijos, se intenta la asignación aleatoria *k* veces. Luego de esto si el problema continúa, el hijo se desecha. Si termina con éxito el proceso de Recombinación, se compara el fitness de ambos hijos y se conserva el mejor.

Ro/Eq	1	2	3	4	5	6	Ro/Eq	1	2	3	4	5	6
1/6	@2	1	5	6	@3	@4	1/6	6	5	@4	3	@2	@1
2/7	3	@4	@1	2	6	@5	2/7	@5	@3	2	@6	1	4
3/8	@4	5	6	@1	@2	3	3/8	@4	6	@5	1	3	@2
4/9	5	6	@4	3	@1	@2	4/9	@3	@4	1	2	@6	5
5/10	@6	@3	2	@5	4	1	5/10	2	@1	@6	@5	4	3
			PADRE X							PADRE Y			

Figura 4.7.- Padres seleccionados para la Recombinación

Ro/Eq	1	2	3	4	5	6	Ro/Eq	1	2	3	4	5	6
1/6							1/6						
2/7							2/7						
3/8	@4			@1			3/8	@4			1		
4/9							4/9						
5/10				@5	4		5/10				@5	4	
			HIJO XY							HIJO YX			

Figura 4.8.- Combinación de filas entre los padres.

En la *Figura 4.8* se muestra a los hijos *XY* y *YX* donde las celdas que contienen un valor corresponden a los genes repetidos en los padres. Luego se completan las celdas vacías aleatoriamente.

Ro/Eq	1	2	3	4	5	6
1/6	@2	1	5	6	@3	@4
2/7	5	@6	4	@3	@1	2
3/8	@4	5	6	@1	@2	3
4/9	@3	@4	1	2	@6	5
5/10	@6	@3	2	@5	4	1

Figura 4.9.- Reparación Completa del Hijo *X*

4.6.- PARAMETROS DEL ALGORITMO

Los últimos detalles que son necesarios mencionar, son los referentes a los parámetros que usará el algoritmo durante su ejecución.

a. *Tamaño de la Población*

Para problemas como el TTP o el MTTP, donde existen muchas instancias para las que se han publicado resultados, es muy importante definir cuantos agentes se generarán y se mantendrán en la población. Existen instancias del problema donde se toman Fixtures de 4 equipos, esto hace que el espacio de búsqueda sea pequeño y baste con unos pocos agentes para lograr una solución. Los trabajos realizados no utilizan búsquedas poblacionales, por lo que no hay una referencia de la cantidad necesaria de agentes. Por esta razón, la cantidad se definirá en el futuro después de la implementación, esto es, como un estudio de ensayo y error [15,24].

b. *Cantidad de Padres que se utilizarán en la Recombinación*

La teoría Memética habla de que se pueden utilizar dos o más padres para crear uno o varios hijos. Pero la recomendación, y lo más usual es que se tomen dos padres para crear dos hijos [20].

c. *Paso Generacional (Reemplazo)*

El paso generacional, es decir la selección de los agentes que morirán para dar paso a los "buenos hijos", se hará simplemente sustituyendo a los agentes con menor beneficio en la población actual. Esto producirá que la población evolucione de manera que todos los agentes sean más prometedores [17].

d. *Estrategia de Selección*

La estrategia de selección de padres será la por torneo, es decir, se tomará un subconjunto de la población de manera aleatoria y se elegirán a los dos mejores para ser padres. Esta estrategia dará opción a los agentes que no tienen un beneficio muy elevado para lograr ser padres [17].

e. *Criterio de Parada*

El criterio de parada será contar una determinada cantidad de ciclos sin obtener una mejora en la calidad de los agentes. A partir de esa información se puede deducir que se ha encontrado una solución. La cantidad específica se determinará en el futuro por la misma razón que se expuso en el tamaño de la población [15,3].

4.7.- MODELO DE SOLUCION

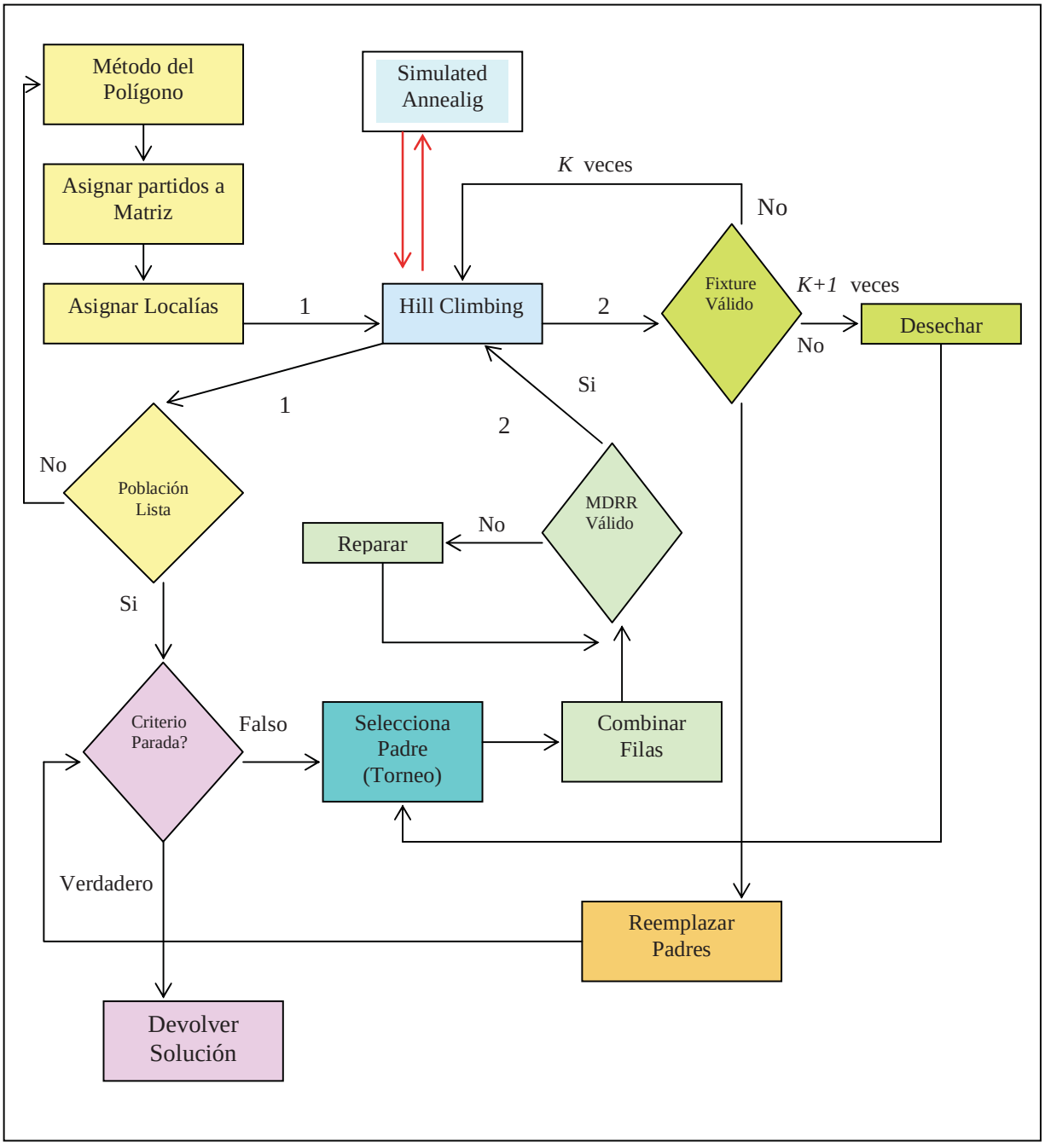


Figura 4.10.- Modelo de la Solución

CAPITULO 5.- IMPLEMENTACION

5.1.- REPRESENTACION DE LAS SOLUCIONES

La implementación de la matriz M*r,e*, se lleva a cabo mediante la utilización de arreglos dinámicos, es decir, que pueden cambiar sus dimensiones. Esto permite ejecutar todas las instancias del problema con el mismo código, solo necesitando el ingreso de la dimensión del la matriz, lo que se hace cuando se ingresa la cantidad de equipos que contendrá el fixture. La manera en que C trabaja matrices dinámicas se define a continuación.

Se define un puntero a puntero con dimensiones 2x*(n-1)* para las filas. Cada celda será un puntero a un arreglo de dimensión *n*. Las celdas se crean pidiendo memoria con la instrucción "malloc" o "calloc".

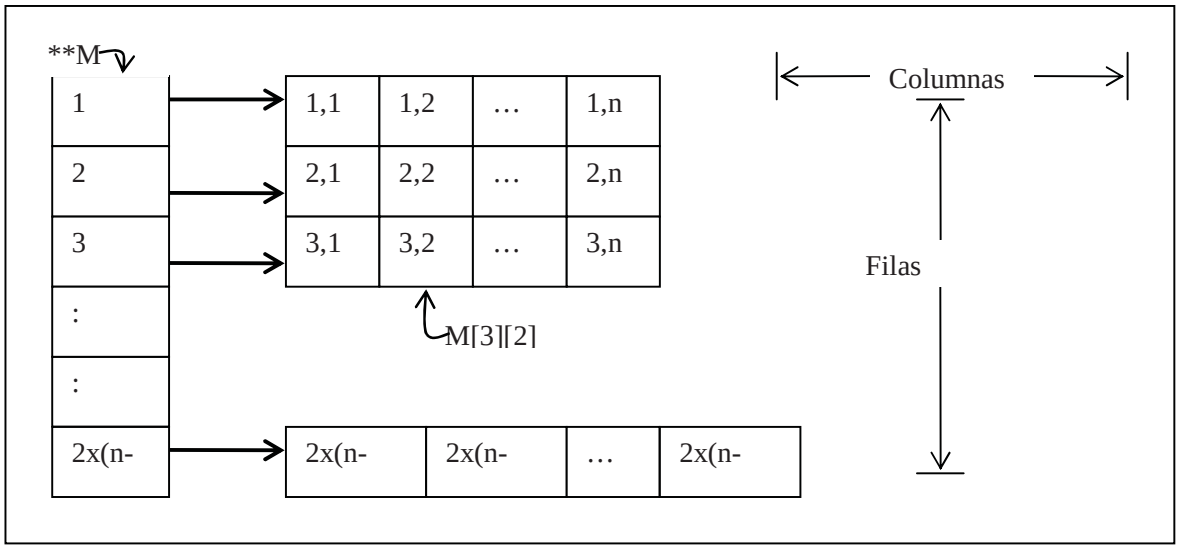


Figura 5.1.- Arreglo dinámico para el Fixture

La *Figura 5.1* muestra la estructura de la matriz M*r,e* en memoria, donde "***M*" corresponde a un puntero a puntero que apunta a un arreglo de punteros que a su vez hace referencia a un arreglo de enteros (en el caso de fixtures). Luego de esta definición se puede

utilizar la estructura como si fuera una matriz ordinaria, es decir, haciendo referencia a sus celdas de la forma "*M[fila][columna]*".

5.2.- FUNCION OBJETIVO

El primer punto que se debe considerar a la hora de resolver un problema, es la razón fundamental por la que se está diseñando un algoritmo. En este caso lo que se persigue es minimizar una función que mide los kilómetros recorridos por los equipos durante el desarrollo de un MDRR. Además dicha función nos servirá como criterio de comparación entre dos o más soluciones.

Se define D como la matriz o cuadro de distancias que existe entre los *n* equipos involucrados en el torneo. D también será una matriz dinámica puesto que depende de la instancia que estemos tratando, es así como las distancias se obtienen de un archivo .txt general para problemas de la familia TTP, donde se encuentran las distancias entre los 16 equipos que componen la instancia mayor. Mediante una función se rescatan las distancias necesarias para cada instancia. La *Tabla 5.1* muestra el cuadro de distancias para la instancia NL8.

-------	Atl	Nym	Phi	Mon	Fla	Pit	Cin	Chi
Atl	0	745	665	929	605	521	370	587
Nym	745	0	80	337	1090	315	567	712
Phi	665	80	0	380	1020	257	501	664
Mon	929	337	380	0	1380	408	622	646
Fla	605	1090	1020	1380	0	1010	957	1190
Pit	521	315	257	408	1010	0	253	410
Cin	370	567	501	622	957	253	0	250
Chi	587	712	664	646	1190	410	250	0

Tabla 5.1.- Cuadro de distancia para NL8.

Parámetros de Entrada: Cantidad de Equipos, Cuadro de Distancias, Matriz M*re.*
Salida: un entero que representa el recorrido total.

5.3- GENERACION DE LA POBLACION INICIAL

Según la *Figura 2.5,* el primer paso del AM es el proceso de inicialización de la población de Agentes. Esta inicialización consiste en la creación de las configuraciones (individuos) y su posterior optimización local (Agentes). Este proceso se divide en 4 etapas:

1.-Se deben determinar los partidos que se jugarán por ronda mediante el Método del Polígono.
2.- Se guardarán los partidos obtenidos en la Matriz M*r,e* .
3.- Se asignarán las localías siguiendo un patrón que respeta las restricciones del MTTP.
4.- Se aplicará un optimizador local a las configuraciones para crear Agentes.

Para la construcción de un individuo se necesita una estructura de datos que soporte la matriz M*r,e* ya definida, y un entero que guarde el fitness asociado a dicha matriz. Además para construir la población es necesario tener acceso fácil a todos los individuos.

Para esto se define una estructura de nodo con el fin de crear una lista enlazada simple que representará la población de agentes a evolucionar.

Figura 5.2.- Estructura de un Individuo

```
struct agente{
  struct agente *siguiente;
  int **matriz;
  int fitness;
};
```

La *Figura 5.2* muestra la estructura de nodo utilizada para representar a un agente. Como se dijo anteriormente "***m*" es la matriz que soporta la estructura del fixture en memoria, "*fitness"* es un entero que guarda la calidad del fixture asociado y "**siguiente"* es un puntero al siguiente nodo o agente en la población.

La lista enlazada o población se encontrará ordenada en todo momento por fitness.

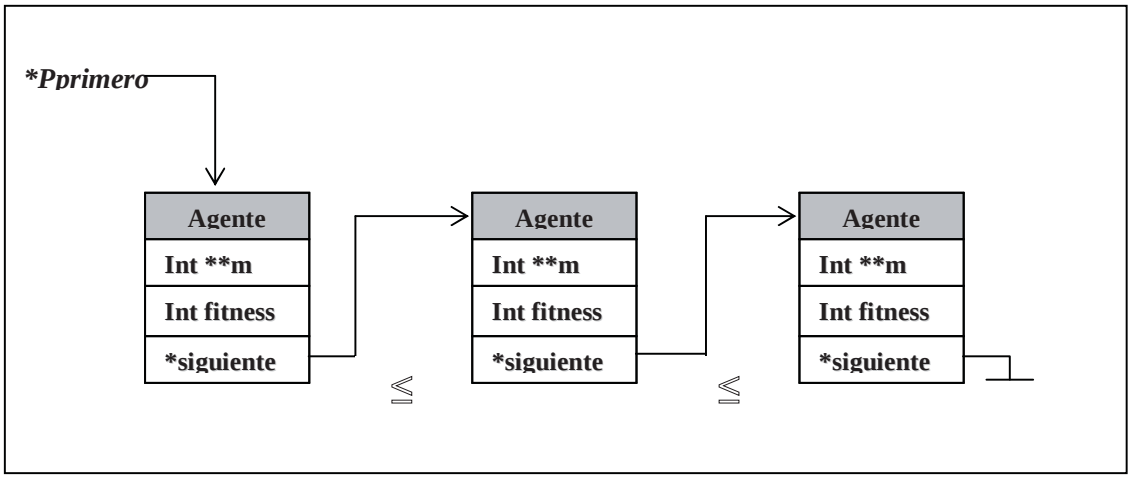
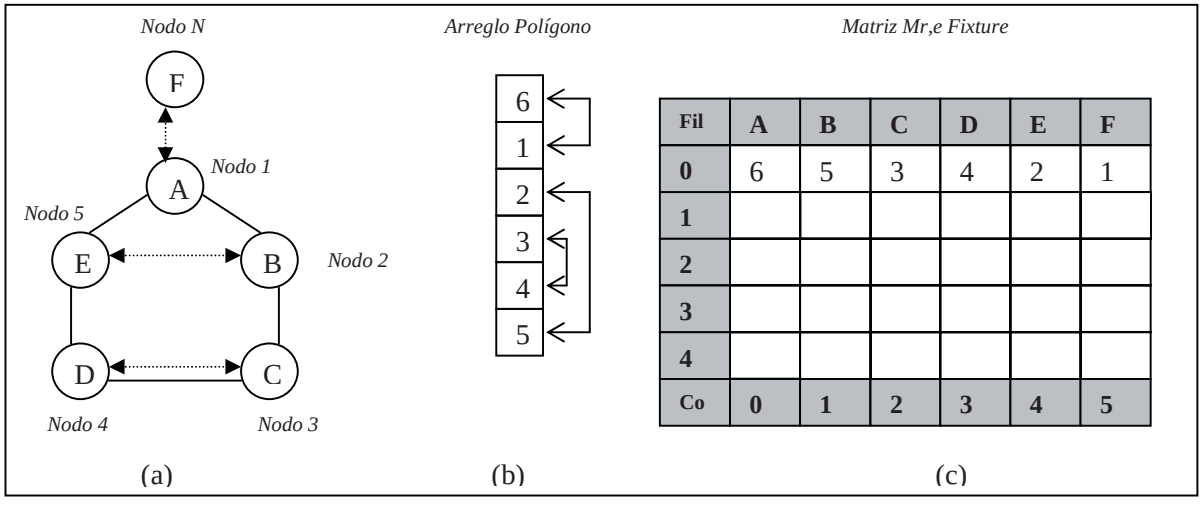


Figura 5.3.- Lista Enlazada o Población

5.3.1.- Método Del Polígono

Parámetros de Entrada : Cantidad de equipos, Matriz M*r,e*, estructura de Polígono.

Estructuras a Utilizar : Estructura de arreglo que representa el polígono, estructura de datos que representa el Fixture (M*r,e)*.



Fil	A	B	C	D	E	F
0	6	5	3	4	2	1
1						
2						
3						
4						
Co	0	1	2	3	4	5

Figura 5.4.- Fase de implementación del polígono y asignación a la matriz M*r,e*, en NL6.

La *Figura 5.4.a* muestra la figura poligonal donde los vértices son los equipos que intervienen en el torneo. En la *Figura 5.4.b* se presenta el arreglo que representa el polígono en el algoritmo, la primera celda (0) corresponde al *Nodo N*, luego las celdas 1...N-1 corresponden a los *Nodos* 1...N-1. El giro del polígono no toma en cuenta el nodo cero, por lo que se simula el giro real y se generan los partidos sucesivos siguiendo la correspondencia mostrada en la figura. La *Figura 5.4.c* muestra la matriz M*r,e* que es la estructura que soportará el Fixture. Las filas están numeradas de 0 a 4 (5 filas), lo que representan las (n-1) rondas en un Fixture NL6. Las columnas están numeradas de 0 a 5 (6 columnas) y representan a los 6 equipos en NL6. El partido **A-F** se encuentra en *M[0][0]*, la forma de llegar a **F-A** en la misma fila es *M[0][M[0][0]-1]*, de igual manera se tratan los demás partidos.

5.3.2.- Asignar Localías

Parámetros de Entrada: Cantidad de Equipos, Matriz M*r,e*.
Código Fuente: Se muestra la parte de discriminación de localía y los casos correspondientes.

Figura 5.5.- Asignación de Localías

```
Procedimiento IPR( int equipos, int **m)
{ int i,j,nx,ny;
      Para i ← 0 hasta (equipos-1)
      Para j ← 0  hasta equipos
         E1← tomar_equipo(M(i,j));
          E2← tomar_rival(E1);
          Nx← calcular(E1);
          Ny←calcular(E2);
            Si (nx =ny AND M(i-1,j)>0)
                asignar_localía(E1)
            Else Si ( M(i-1,j)<0)  asignar_localía(E2);
            Else asignar_localía(aleatorio);
            Fin Si
            Si (nx>ny AND  M(i-1,j)>0) asignar_localía(E1);
            Else asignar_localía(E2);
           Fin Si
  Fin Para
Fin Para
}Fin
```

La *Figura 5.5* muestra el proceso de asignar localías a los partidos insertos en la matriz. Se discrimina la asignación de la localía según dos valores que son calculados a los dos equipos involucrados en un partido dado (*nx* y *ny*). Estos valores representan la cantidad de partidos que han jugado de forma consecutiva de local o visita dichos equipos. Luego según el caso se asigna la localía, si *nx* y *ny* son iguales, se asigna aleatóriamente la localía igual que a la primera ronda [21]. El proceso anterior se realiza por cada par de equipos que se enfrentan en un partido, afectando directamente el partido de revancha, es decir, si el partido que se está evaluando se encuentra en la fila "i", se asignará la localía inversa al mismo partido en la fila "(i+equipos-1)" de la matriz M*r,e*.

El seguir lo anterior entrega como resultado un Fixture Mirrored Double Round Robin (MDRR). Es posible que la salida sea un Fixture infactible, esto según se verá no es un problema ya que la infactibilidad pasa por el quebrantamiento de la restricción impuesta por U=3, lo que puede ser resuelto mediante un movimiento de filas o cambio de localía. Estos movimientos son aplicados en la fase de búsqueda local que se debe efectuar cada vez que se genera una solución mediante el proceso antes descrito.

Existe un problema en la asignación de localías, puesto que si bien para un SRR no se quebranta la restricción de la variable U=3, al duplicar el Fixture puede que si ocurra. Supongamos que el equipo A juega el primer partido de local, y los tres últimos de la primera fase en calidad de visita. Al duplicar el Fixture se tiene que el equipo A juega el primer partido de revancha en calidad de visita, sumando en total 4 partidos consecutivos en esa condición superando el límite de U=3.

No existe un método de reparación totalmente efectivo, pero las fallas en la asignación de localías pueden repararse en la fase de búsqueda local. Lamentablemente para Fixtures superiores a NL10 la probabilidad de generar Fixtures factibles es muy baja, lo que puede hacer fracasar la generación del agente si la búsqueda local no puede reparar el Fixture. Por esta razón, antes de pasar un Fixture infactible por la búsqueda local, se le aplica un movimiento determinado que puede ayudar a reparar el Fixture más rápido.

Al ver la alta dependencia que tenía el error de la primera y última fila del fixture SRR, se decidió intercambiarlas. Así puede que el error desparezca, o bien pueda recuperarse con pocos movimientos más.

5.4.- OPTIMIZADOR LOCAL

La estrategia de búsqueda local es la que representa el aprendizaje de los individuos para convertirse en agentes. Esto lo hace a través de la exploración de vecindarios de las soluciones generadas inicialmente, o los hijos que salgan de la recombinación de 2 o más soluciones padres. También el optimizador local puede verse como un meta-operador de mutación, cambiando aleatóriamente la configuración de la solución de manera iterativa o como lo determine la Heurística [10].

Los vecindarios dependen de las transformaciones aplicadas como se dijo anteriormente. Por esta razón la existencia de varias transformaciones nos permitirá diversificar la población ya que se explorará en diferentes direcciones del espacio [15]:

5.4.1.- Intercambio Parcial De Rondas

La implementación de este movimiento se explica más claramente en la *Figura 5.6,* donde se muestra el funcionamiento de la codificación realmente hecha. La codificación se muestra como pseudo-código por la extensión de la función real en la *Figura 5.7.*

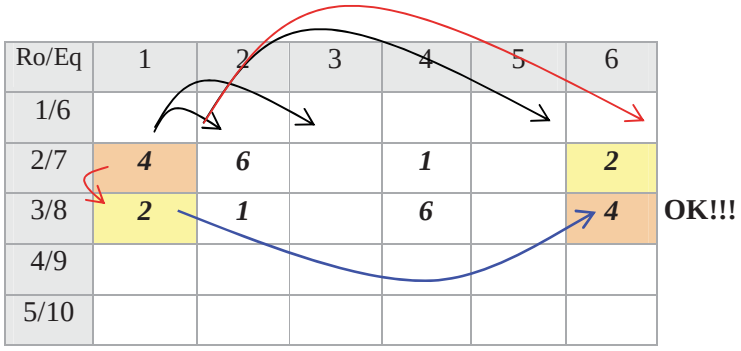


Figura 5.6.- Funcionamiento de IPR

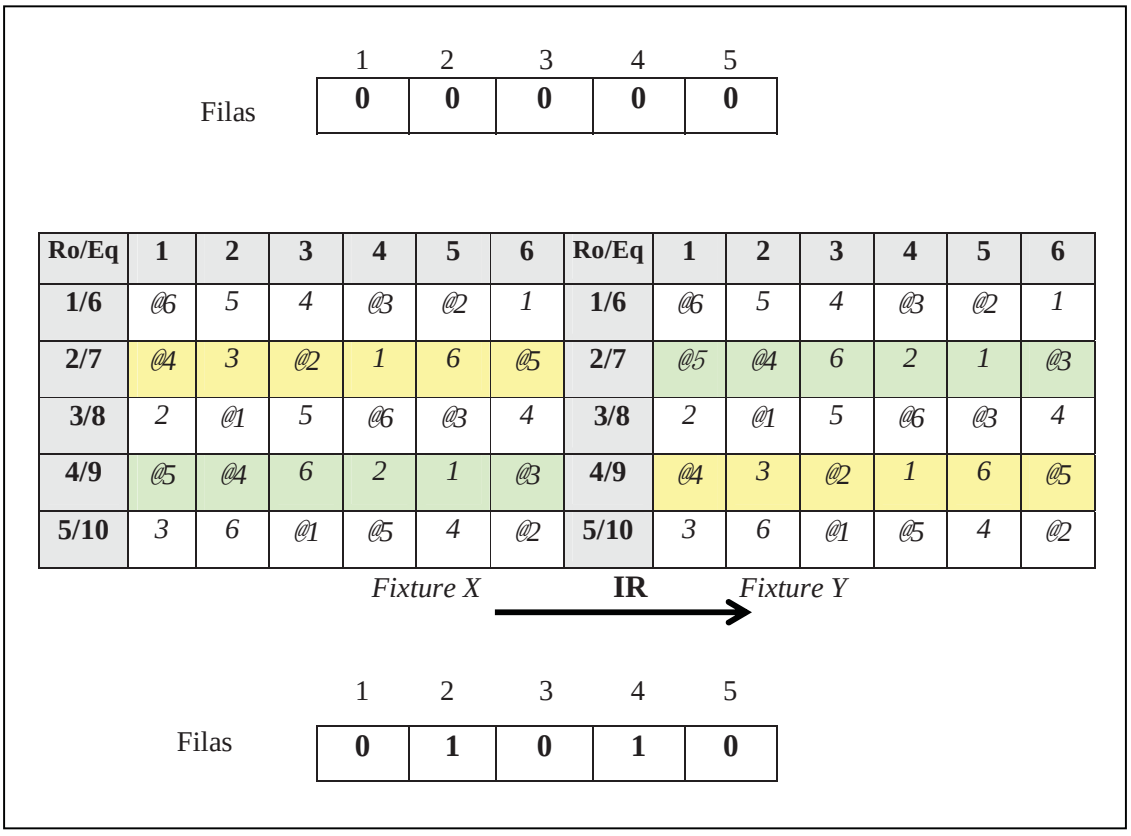
Las flechas negras indican la búsqueda de un partido diferente al actual (4-1), cada vez que se encuentre una celda de un partido diferente, comienza la búsqueda del valor contenido en la celda a lo largo de la columna de la celda actual. Cuando se encuentra el valor (flechas rojas), se revisa la columna de la celda encontrada como partido diferente en la fila donde se encontró la coincidencia (flechas azules), si es igual, el movimiento puede realizarse. En este caso la celda actual es la (2,1) que contiene un "4", la celda del partido diferente es la (2,6) que contiene un "2". Se busca un "2" en la columna de la celda actual encontrándose en la celda (3,1). Luego se revisa en la columna del "2" y fila del "2" encontrado, si es un "4", el movimiento puede realizarse para evaluar si el Fixture resultante es mejor que el actual. Si es así se conserva el cambio, sino se rechaza.

Figura 5.7.- Procedimiento IPR

```
Procedimiento IPR( int equipos, int **m, int**distancia)
{ int i,j,y,x;
  int ** fixture_auxiliar;

    Para i ← 0 hasta (equipos-1)
     Para j ← 0  hasta equipos

        Para y← i  hasta (equipos-1)
          Para x ← j  hasta equipos

            Si ( m(i,j) = m(y,x)  and m(i,x) = m(y,j)  ) entonces
                 Fixture_auxiliar← IPR(m);
                 Si ( fitness(fixture_aux) < fitness(m) )
                       m←fixture_auxiliar;
                  Sino
                       fixture_auxiliar← m;
                  Fin Si
              Fin Si
          Fin Para
         Fin Para
   Fin Para
  Fin Para
} Fin Procedimiento
```

5.4.2.- Intercambio De Rondas

Este movimiento consiste en seleccionar dos rondas *r1* y *r2*, e intercambiarlas con el fin de encontrar un Fixture con un fitness mejor. Se puede ver que al aplicar esta transformación no se viola ninguna restricción relacionada con el MDRR (*3.1, 3.2, 3.3*), pero si las referentes al MTTP (*3.4, 3.5, 3.6*).



Ro/Eq	1	2	3	4	5	6	Ro/Eq	1	2	3	4	5	6
1/6	@6	5	4	@3	@2	1	**1/6**	@6	5	4	@3	@2	1
2/7	@4	3	@2	1	6	@5	**2/7**	@5	@4	6	2	1	@3
3/8	2	@1	5	@6	@3	4	**3/8**	2	@1	5	@6	@3	4
4/9	@5	@4	6	2	1	@3	**4/9**	@4	3	@2	1	6	@5
5/10	3	6	@1	@5	4	@2	**5/10**	3	6	@1	@5	4	@2

Figura 5.8.- Intercambio de Rondas IR

En la *Figura 5.8* se ve el intercambio de las rondas 2 y 4 en el Fixture Y (auxiliar), luego este se compara con el Fixture X y si su calidad es mejor se conserva, sino se desecha. Si se conserva la estructura vector fijará dichas filas con el fin de no repetir el intercambio, cambiando el valor en el vector "filas" de "0" a "1". Esto produce que se reduzca el tiempo para todo este proceso.

5.4.3.- Intercambio De Localías

La vecindad Intercambio de Localía IL(*x*) de un Fixture $x \in E$ consiste en todos los Fixtures que se pueden obtener a partir de *x,* cambiando la localía de los dos partidos entre un mismo par de equipos *e1* y *e2.* Es decir, que si en *x, e1* jugaba de local contra *e2* en la ronda *r1,* y de visitante en la ronda *r2,* las únicas diferencias entre *x* y el vecino que se obtiene permutando las localías de los equipos *e1* y *e2* son simplemente que en este último, *e1* juega contra *e2* de visitante en la ronda *r1* y de local en la ronda *r2.* Esto implica cambiar el signo de los valores de las celdas M[*r1,e1*], M[*r2,e1*], M[*r1,e2*], M[*r2,e2*]. Una permutación de este tipo sólo puede violar las restricciones 3.*5* y 3.*6.* [15].

Ro/Eq	1	2	3	4	5	6
1/6	@6	5	4	@3	@2	1
2/7	@4	3	@2	1	6	@5
3/8	2	@1	5	@6	@3	4
4/9	@5	@4	6	2	1	@3
5/10	3	6	@1	@5	4	@2

IL ↓

Ro/Eq	1	2	3	4	5	6
1/6	@6	5	4	@3	@2	1
2/7	4	3	@2	@1	6	@5
3/8	2	@1	5	@6	@3	4
4/9	@5	@4	6	2	1	@3
5/10	3	6	@1	@5	4	@2

Figura 5.9.- Intercambio de Localías

En la *Figura 5.10* se muestra la estructura de Fixture que se utiliza para fijar los partidos cuyas localías fueron intercambiadas. Esta estructura se inicializa con valores “0” para todas sus celdas y se cambian a “1” a medida que se encuentran mejoras en los Fixtures.

Ro/Eq	1	2	3	4	5	6
1/6	*0*	*0*	*0*	*0*	*0*	*0*
2/7	*1*	*0*	*0*	*1*	*0*	*0*
3/8	*0*	*0*	*0*	*0*	*0*	*0*
4/9	*0*	*0*	*0*	*0*	*0*	*0*
5/10	*0*	*0*	*0*	*0*	*0*	*0*

Figura 5.10.- Estructura para fijar localías

Figura 5.11.- Procedimiento IL

```
Procedimiento IL( int equipos, int **m, int**distancia)
{ int i, j, anterior;
  int ** fixture_auxiliar;

  Para i ← 0 hasta (equipos -1)
      Para j ← 0  hasta equipos

         Anterior ← fitness(m);
         m ← IL(m,i ,j) ;

         Si (fitness(m) < Anterior ) entonces
            Fixture_auxiliar ← fijar_celdas(i,j);
            i,j ← 0 ;
         Sino
            m ← deshacer_cambio(i,j);
         Fin Si
      Fin Para
  Fin Para
} Fin Procedimiento
```

La *Figura 5.11* muestra el funcionamiento de la búsqueda por intercambio de localías, donde se prueba con todos los cambios posibles uno a uno de los Fixtures que arrojen mejoras, si el movimiento no arroja mejoras se deshace con el proceso inverso de multiplicación de las celdas involucradas.

Nota : En todos los ejemplos de vecindarios sólo se muestra un SRR. Como el MTTP es para un MDRR, estos cambios también deben verse reflejados en la segunda vuelta de partidos, lo que no afecta la factibilidad de la solución.

5.5.- SELECCION

El proceso de selección consiste en encontrar de entre la población, aquellas configuraciones que serán los padres de la nueva generación. Existen diferentes formas de selección, pero de acuerdo a lo establecido en la fase de investigación, este proceso se realizará utilizando la técnica de selección por torneo.

La selección por torneo consiste en tomar un sub-grupo de agentes de la población del que se elegirán los dos agentes con mejor calidad. La cardinalidad de este sub-grupo es muy importante ya que debe encontrarse un equilibrio entre el costo de comparación de una cantidad grande de agentes, y la probabilidad de no cubrir los mejores agentes con una cardinalidad pequeña.

En realidad seleccionar siempre a los mejores agentes como padres puede causar una convergencia prematura a un óptimo local. Por otro lado puede que se gane en diversidad de la población, si se elige un padre con una calidad menor y así explorar espacios no cubiertos en la búsqueda.

Parámetros de Entrada : Puntero al primer Agente de la población, cantidad de agentes en la población, cardinalidad del sub-grupo.

Salida : Puntero a la estructura definida Padres, cuyos datos son los dos mejores padres del sub-grupo.

El proceso define un arreglo dinámico con la cantidad de agentes que existen en la población, con el fin de llevar el registro de cuales ya fueron seleccionados y no repetir procesos inútilmente. Luego se debe ingresar la cardinalidad del sub-grupo (2...cantidad), la que servirá como criterio de parada para el proceso siguiente.

Se itera tantas veces como indique la cantidad, se genera un número aleatorio entre 1 y la cantidad de agentes con el fin de recorrer la población hasta llegar a ese número de agente. Se seleccionan agentes y cada vez se van comparando con los ya obtenidos, conservando los dos mejores cada vez. Para no repetir el agente seleccionado, cada vez que se elige un agente, se asigna a la casilla correspondiente en el arreglo creado un 1. Luego de terminar de iterar, se garantiza que en la estructura "Padres" estarán los dos mejores agentes del subgrupo.

5.6.- RECOMBINACION

El operador de Recombinación es el encargado de crear agentes a partir de dos o más agentes llamados padres. En un AM este operador es inteligente, es decir, es propio del problema. En el caso del MTTP es bastante complicado usar cualquier operador conocido de crossover, ya que los hijos generados pueden violar de manera severa las restricciones tanto del MDRR como del TTP.

La función de Recombinación devuelve un puntero al agente creado luego del cruce de los padres. Los parámetros de entrada son los dos padres, la cantidad de equipos y el cuadro de distancia. Las estructuras necesarias para realizar el proceso son dos Fixtures auxiliares que representarán los dos hijos a crear. De esta forma uno de los Fixtures creados deberá ser destruido o liberado con el fin de no acaparar memoria inútilmente.

La función "completar_hijo" trata de asignar valor a todas las celdas de los hijos cuya condición sea "0" (vacía). Dentro de esta función existe un proceso de revisión de fallos, es decir, que si el Fixture no es completado existe la posibilidad de intentar su relleno una cantidad *k* de veces. Si luego de esto no se ha completado el fixture, se desecha y asigna un fitness alto para que no pueda competir con el otro hijo.

En el caso que ambos hijos fracasen, unos de ellos se insertará en la población con un fitness muy alto, lo que desencadenará su muerte inmediata.

Figura 5.12.- Proceso de Recombinación

```
Procedimiento Recombinacion(struct agente padre1, struct agente padre2)
{ int i, j;
  int ** Hijo1,Hijo2;
    Para i ← 0 hasta (equipos -1)
          Para j ← 0  hasta equipos
             Si ( padre1(i,j) = padre2(i,j) ) entonces
                 Hijo1(i,j), Hijo2(i,j) ← padre1(i,j);
             Fin Si
          Fin Para
  Fin Para
           Hijo1 ← completar_hijo(hijo1);
           Hijo2 ← completar_hijo(hijo2):
  Si (fitness(Hijo1) < fitness(Hijo2) ) entonces
          Liberar(Hijo2);
         Devolver(Hijo1);
  Sino
          Liberar(Hijo1);
          Devolver(Hijo2);
  Fin Si

} Fin Procedimiento
```

5.7.- REEMPLAZO

La técnica de reemplazo se refiere a la forma en que se renovará la población de Agentes. Esta técnica se llevará a cabo de forma en que el hijo generado se compare con la población actual y se inserte en orden de calidad. Esto implica que siempre se eliminará el último nodo, ya que la lista enlazada que representa la población se encuentra ordenada según lo dicho en la *Sección 5.3.* La única forma que el hijo no se inserte en la población es que tenga peor calidad que todos los Agentes en la población.

5.8.- COMPONENTE SIMULATED ANNEALING (SA)

Con el objetivo de lograr entregar soluciones que se acerquen lo más posible a los mejores publicados para el MTTP, es necesario tener alternativas de código que se pueda utilizar para sustituir o complementar código existente. Es por esto que se ha implementado una variante de búsqueda local que sea más potente, y que pueda encontrar mejores soluciones. La heurística

codificada recibe el nombre de Templado Simulado (Simulated Annealing), la cual será explicada a continuación.

5.8.1.- Marco Teórico

Este procedimiento se basa en una analogía con el comportamiento de un sistema físico al someterlo a un baño de agua caliente. Simulated Annealing (SA) ha sido probado con éxito en numerosos problemas de optimización, mostrando gran "habilidad" para evitar quedar atrapado en óptimos locales. Debido a su sencillez de implementación así como a los buenos resultados que iban apareciendo, experimentó un gran auge en la década de los 80 [11].

Las leyes de la termodinámica establecen que, a una temperatura *t*, la probabilidad de un aumento de energía de magnitud ∂E viene dada por la expresión siguiente:

$$P(\partial E) = e^{\frac{-\partial E}{kt}} \tag{5.1}$$

Donde *k* es la constante de Boltzmann.

Se pensó que se podía hacer una analogía entre los parámetros que intervienen en la simulación termodinámica y los que aparecen en los métodos de optimización local. Para ello, establecen un paralelismo entre el proceso de las moléculas de una sustancia que van colocándose en los diferentes niveles energéticos buscando un equilibrio, y las soluciones visitadas por un procedimiento de búsqueda local. Así pues, SA es un procedimiento basado en búsqueda local en donde todo movimiento de mejora es aceptado y se permiten movimientos de no mejora de acuerdo con unas probabilidades. Dichas probabilidades están basadas en la analogía con el proceso físico de enfriamiento y se obtienen como función de la temperatura del sistema.

La estrategia de SA es comenzar con una temperatura inicial *alta*, lo cual proporciona una probabilidad también alta de aceptar un movimiento de no mejora. En cada iteración se va reduciendo la temperatura y por lo tanto las probabilidades son cada vez mas pequeñas conforme avanza el procedimiento y nos acercamos a la solución óptima. De este modo, inicialmente se realiza una diversificación de la búsqueda sin controlar demasiado el coste de las soluciones

visitadas. En iteraciones posteriores resulta cada vez más difícil el aceptar malos movimientos, y por lo tanto se produce un descenso en el coste [11].

En otras palabras, pensando en el MTTP se puede utilizar la *Ecuación 12.4* como sigue:

$$P = exp^{\frac{-(cost(S2)-cost(S1))}{T}} \quad (5.2)$$

Donde P representa la probabilidad de aceptación, *cost(S1)* denota la solución actual, *cost(S2)* denota la solución siguiente y *T* es la temperatura [24].

Aplicando la teoría al problema, para el TTP se recomienda utilizar un a temperatura inicial *T=400,* la cual debe decrecer muy lentamente. La razón propuesta es *T'= rT,* con $r = 0.99$.

DISEÑO

Para el diseño del componente SA se utilizarán los movimientos definidos. El inconveniente de esto es que SA plantea que la temperatura debe decrecer lentamente hasta un valor muy pequeño para poder filtrar de mejor forma las mejores soluciones hacia el final del proceso. Esto significa que se necesita realizar una cantidad de movimientos significativo para lograr lo anterior.

Dentro de las transformaciones definidas, IPR no es posible realizarla muchas veces en un fixture, al igual que IR. El único movimiento al que es posible incorporar SA es el intercambio de localías, ya que existen muchos partidos dentro de un fixture, sobretodo para las instancias más grandes.

Por ende la estrategia será diferente que en Hill Climbing, ya que no se harán los movimientos secuencialmente sino que cada vez que se acepte un movimiento, el proceso de IL llamará a los otros dos movimientos combinando sus bondades para lograr encontrar óptimos nuevos en lugares no explorados.

La aplicación de SA como complemento o sustitución se determinará luego de las diferentes pruebas de componentes al algoritmo. Por ahora se adelanta que la idea es la sustitución

en la primera fase de búsqueda local y el complemento en la segunda fase, ya que Hill Climbing ha demostrado ser ineficiente para superar los óptimos locales encontrados en la primera fase, sobretodo para las instancias mayores.

IMPLEMENTACION

Respetando lo explicado en el diseño, es claro que la inserción del componente modifica el código presentado en la *Figura 5.11,* de forma de discriminar la aceptación de una solución de no mejora según la probabilidad mostrada en la *Ecuación 5.2*.

Figura 5.13.- IL con componente SA

```
Procedimiento IL( int equipos, int **m, int**distancia)
{ int i, j, anterior;
  int ** fixture_auxiliar;
  float Probabilidad;
  float T=400;

 Para i ← 0 hasta (equipos -1)
  Para j ← 0  hasta equipos
          Anterior ← fitness(m);
          m ← IL(m,i ,j) ;
          Si (fitness(m) < Anterior ) entonces
              Fixture_auxiliar ← fijar_celdas(i,j);
              i,j ← 0 ;
          Sino
              Probabilidad ← Aleatorio[0..1];
              Si (Probabilidad < SA(T, cost(S1), cost(S2)) entonces
                    Fixture_auxiliar ← fijar_celdas(i,j);
                     i,j ← 0 ;
              Sino
                    m ← deshacer_cambio(i,j);
              Fin Si
          Fin Si
          T ← 0.99*T;
   Fin Para
Fin Para
} Fin Procedimiento
```

La *Figura 5.13* difiere con el IL original solo cuando se logra una solución peor como resultado de la aplicación del movimiento. Cuando esto ocurre, un número aleatorio entre 0 y 1 es generado y comparado con el resultado de la función SA que devuelve la probabilidad de aceptación. Si el número generado está dentro de la probabilidad de aceptación, el movimiento se acepta, sino se rechaza. Cada vez que la temperatura caiga, el número devuelto por la función SA será más pequeño y se tendrá menos probabilidad que el número generado supere este valor, con lo que cada vez se irán aceptando menos soluciones que no mejoren la solución actual.

5.9.- ALGORITMO MEMETICO PARA EL MTTP

Luego de revisar cada parte del algoritmo y conocer el modelo de diseño, se está en condiciones de mostrar la codificación final del algoritmo propuesto para tratar el problema del MTTP. En la *Figura 5.14* se muestra el pseudo-código que describe el algoritmo y que respeta el esquema mostrado en la *Figura 2.5.*

Figura 5.14.- Algoritmo Memético para el MTTP

```
Procedure Algoritmo Memetico()
{ Agentes *Poblacion;

        Población ← Crear_Individuos(cantidad_agentes);
        Población ← Optimizar_Localmente(Población);

        Mientras (criterio de parada)
        {       Padres ← Selección_Torneo(Población, cardinalidad);
                Hijo ← Recombinacion(Padres);
                Hijo ← Optimizar_Localmente(Hijo);
                Población ← Reemplazo(Hijo);
        }
        Fin Mientras
}
```

Donde:

- Crear_Individuo: consta del método del polígono y la asignación de Localías.
- Optimizar_Localmente: puede ser Hill-Climbing o Simulated Annealing.
- Reemplazo: Eliminación del peor agentes de la población.

5.10.- ESTRUCTURA DE ARCHIVOS

El algoritmo está separado en archivos los cuales guardan funciones relacionadas que no se han explicado, pero que son utilizadas por muchas funciones o procedimientos vistos. A continuación se revisarán los archivos existentes y la jerarquía implícita en las funciones que guardan.

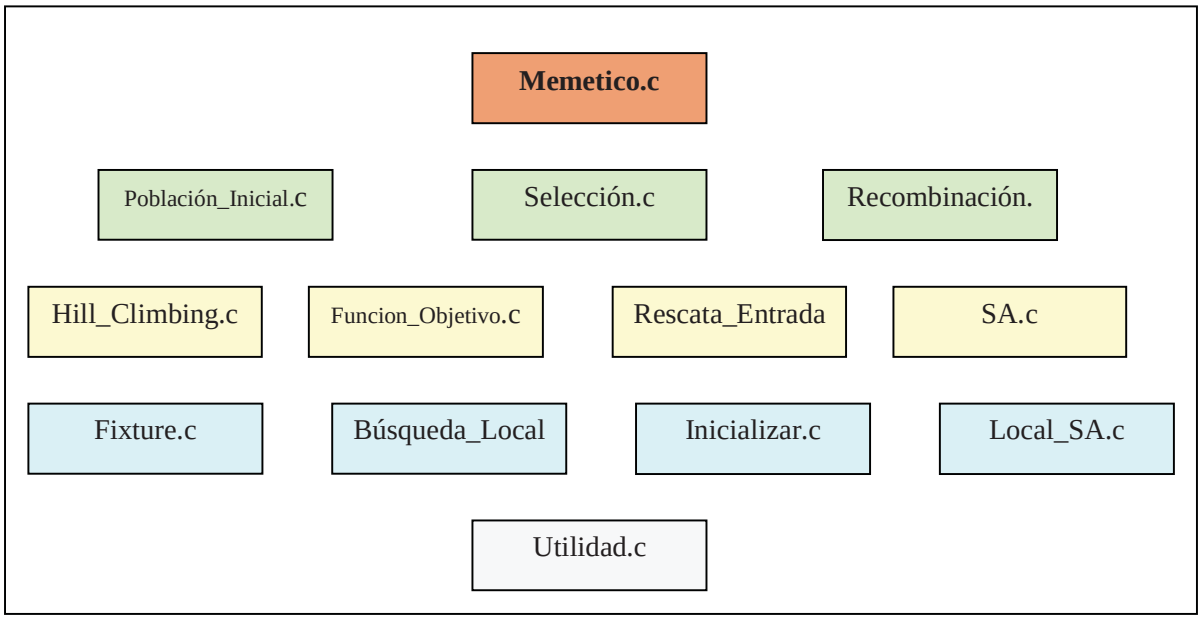


Figura 5.15.- Ordenamiento lógico de los códigos separados

La *Figura 5.15* muestra una separación en capas de los archivos codificados, esta separación dada por colores y representa la especificación del código contenido en los archivos. Mientras más abajo en la figura, más específico es el código. Esto hace que el algoritmo sea más comprensible en las fases superiores y se ordene de manera tal que las funciones contenidas en los niveles inferiores pueden reutilizarse.

La reutilización de código es un factor importante en el sentido de la optimización, ya que reduce el tiempo de ejecución y el peso del algoritmo.

A continuación se detalla la función principal de cada archivo de código. La *Tabla 5.2* contiene el nombre del archivo y su función dentro del algoritmo.

ARCHIVO	*DESCRIPCION*
Memético.c	Archivo principal, contiene el modelo de solución propuesto. Su código es simple por que las funciones que realizan el trabajo son transparentes a él.
Población_Inicial.c	Se encarga de crear un "Agente" e insertarlo en la población.
Selección.c	Se encarga de seleccionar los dos mejores padres desde un sub-grupo de la población.
Recombinación.c	Recombina los dos padres seleccionados y crea un hijo.
Hill_Climbing.c	Es el encargado de aplicar los movimientos a las configuraciones con el fin de encontrar mejoras. Su importancia radica en que se puede cambiar fácilmente la estrategia de aplicación.
Función_Objetivo.c	Es la encargada de evaluar los fixtures y verificar su factibilidad.
Rescata_Entradas.c	Crea el cuadro de distancias acorde a la instancia que se este tratando.
Fixture.c	Contiene las funciones que crean los fixtures, como el método del polígono, etc.
Búsqueda_Local.c	Contiene el código de los movimientos que es posible aplicar.
Inicializar.c	Contiene las funciones para manejar el polígono como estructura de datos.
Utilidad.c	Contiene funciones pequeñas necesarias para ahorrar código.

Tabla 5.2.- Archivos de código fuente y sus funciones.

CAPITULO 6.- PRUEBAS Y RESULTADOS

Luego de haber presentado la definición del problema, la técnica a utilizar, los componentes de la solución, el diseño y la implementación, es hora de verificar si los resultados son los esperados, es decir, si se ha logrado mejorar los mejores resultados obtenidos hasta el momento para la instancia espejada del TTP.

Las pruebas se realizarán de forma de extraer el mayor conocimiento posible acerca del comportamiento de las heurísticas que componen el algoritmo memético, su eficiencia, eficacia y rendimiento en conjunto. En primer lugar se realizarán pruebas de componentes para diferentes instancias, tratando de aislar los parámetros involucrados y determinar valores aproximados buenos para estos, ya que los valores óptimos de los parámetros corresponden a un problema combinatorio que no es el caso tratar en este trabajo.

Luego se realizarán pruebas globales con el fin de combinar los parámetros encontrados y aproximarnos un poco más a soluciones con mejor calidad. Finalmente se realizarán combinaciones con el código del componente SA, tratando de incorporarlo como sustituto o complemento según lo digan los resultados. A continuación se muestran los resultados a los que se pretende llegar o superar.

6.1.- RESULTADOS PUBLICADOS

Los siguientes resultados corresponden al trabajo de Celso Ribeiro y Sebastián Urrutia en Brasil [24]:

Instancia	*Mejor Resultado (Km)*	*Tiempo (Seg.)*
NL4	(*)8276	
NL6	(*)26588	
NL8	(*)41928	3.2
NL10	64024	650.6
NL12	120655	3562.3
NL14	208086	22078.5
NL16	293635	42290.8

Tabla 6.1.- Mejores Resultados

6.2.- PRUEBAS DE COMPONENTES

Las pruebas de componentes tienen el objetivo de optimizar el algoritmo mediante el mejoramiento de las heurísticas subordinadas que lo componen. Así encontrando el mejor funcionamiento de lo que se tiene, se puede globalmente llegar a mejores resultados.

Los componentes se probarán en el orden presentado en la *Figura 2.5*, que muestra la estructura del algoritmo memético. Así la primera fase a probar será la de generación de la población inicial, donde recordemos se incluye una optimización local mediante alguna heurística. Luego se probarán sucesivamente cada movimiento definido por separado, el método de selección y la recombinación

Al terminar cada etapa se concluirá con respecto a los resultados obtenidos, para luego al final dar un resumen de las conclusiones de manera más global.

6.2.1.- Pruebas De Generación De Población Inicial

Para realizar esta prueba se fijarán los parámetros involucrados con el fin que todas las poblaciones generadas, lo hagan bajo las mismas condiciones. Se intenta identificar la diferencia de calidad en los agentes generados bajo los siguientes cambios:

- Generación sin búsqueda Local (Individuos).
- Generación con Hill Climbing.
- Generación con Simulated Annealing.

Las instancias que se analizarán como muestra representativa serán NL8, NL12, NL16. La cantidad de agentes que se generarán serán veinte. El tiempo es un factor clave, así para completar el análisis y dar una conclusión se examinará el tiempo consumido para cada instancia.

a.1)

20 Agentes NL8			
Nº Agente	***s/Búsqueda Local***	***c/Hill Climbing***	***c/SA***
1	51791	49094	45815
2	53361	49615	47604
3	55952	49678	48099
4	56130	49752	48346
5	56516	49832	48484
6	57457	50022	48490
7	57520	50057	48537
8	57835	50558	48856
9	57932	50863	48863
10	58207	51082	49215
11	58604	51745	49254
12	59155	52174	49573
13	59415	52195	49585
14	59960	52806	50166
15	60230	52891	50223
16	60410	53128	50288
17	60867	53324	50529
18	60910	53461	50662
19	61064	54186	51931
20	61415	54635	51936

Tabla 6.2.- Datos Generación de Agentes NL8

a.2)

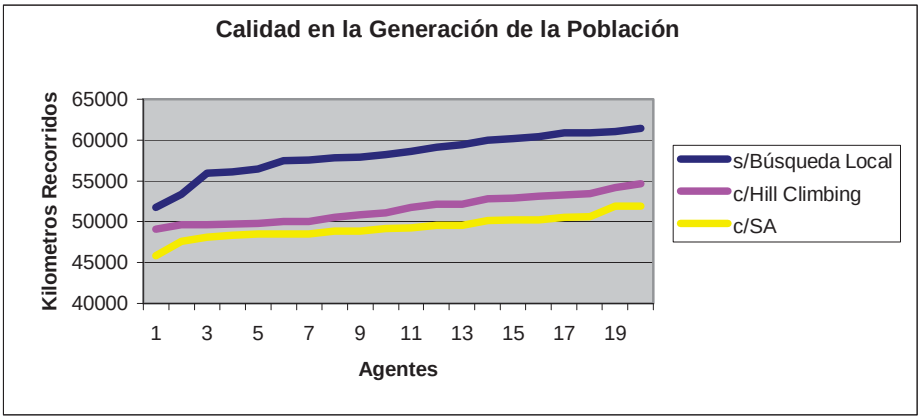


Figura 6.1.- Gráfico Calidad de Población NL8

b.1)

20 Agentes NL12			
Nº Agente	*s/Búsqueda Local*	*c/Hill Climbing*	*c/SA*
1	160055	145763	140000
2	160572	146146	145107
3	164614	147488	146692
4	168099	147520	146892
5	169337	148879	148093
6	172189	150191	148178
7	172264	151207	148569
8	173459	151350	148643
9	174050	152129	149114
10	174284	153349	149898
11	174469	153861	150340
12	174761	155202	150443
13	175073	157112	150805
14	175111	157809	151854
15	175506	158118	151884
16	176103	158230	151903
17	177670	158805	152654
18	177889	162729	152658
19	179764	163330	154381
20	180460	163491	154904

Tabla 6.3.- Datos Generación de Agentes NL12.

b.2)

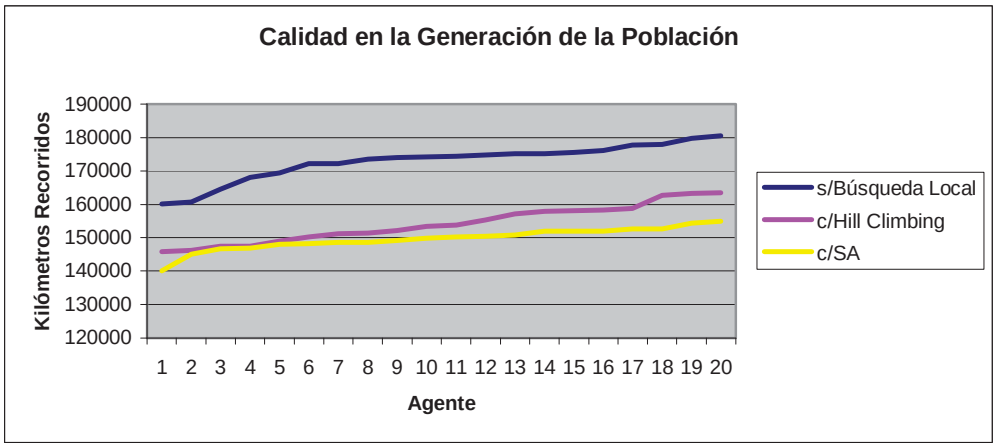


Figura 6.2.- Gráfico Calidad de Población NL12

c.1)

20 Agentes NL16			
Nº Agente	*s/Búsqueda Local*	*c/Hill Climbing*	*c/SA*
1	441383	380329	363350
2	443334	384764	382777
3	452109	386484	390903
4	455394	388627	392140
5	457612	390596	392830
6	457789	393622	394839
7	462992	402317	395440
8	465180	406651	396903
9	467458	406989	398502
10	469654	412771	398714
11	472399	413611	401222
12	473270	416251	404577
13	476036	416582	404928
14	477232	416701	405139
15	478599	418731	406229
16	484479	422631	412344
17	487524	424532	420750
18	489057	427316	422038
19	490268	430155	424512
20	496325	431662	425445

Tabla 6.4.- Datos Generación de Agentes NL16.

c.2)

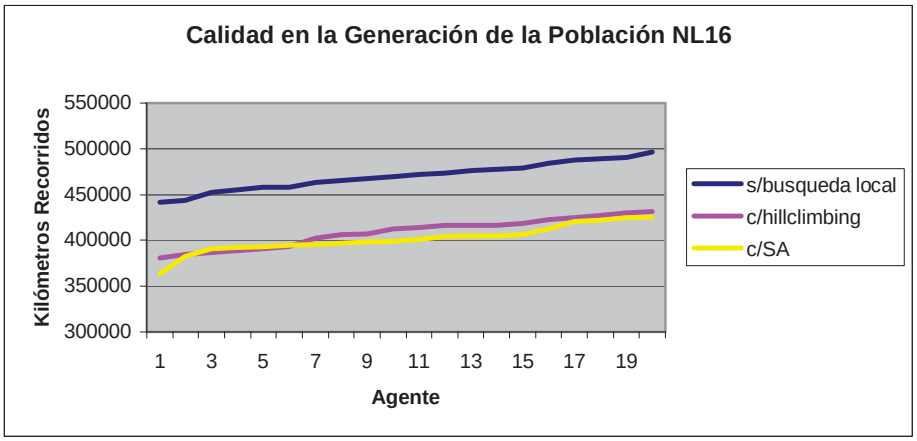


Figura 6.3.- Gráfico Calidad de Población NL16

d) Pruebas de Tiempo consumido en la Generación de la Población Inicial. La ***Tabla 14.5*** muestra los tiempos consumidos en la generación de 50 agentes y la instancia NL10. Se variará el método de búsqueda local para evaluar el proceso completo. La ***Figura 14.4*** grafica esta situación.

	Tiempo en Segundos	
Instancia	***c/Hill Climbing***	***c/SA***
NL4	0	0
NL6	0	0
NL8	0	1
NL10	1	2
NL12	2	3
NL14	4	4
NL16	8	7

Tabla 6.5.- Tiempo en Generación de Población

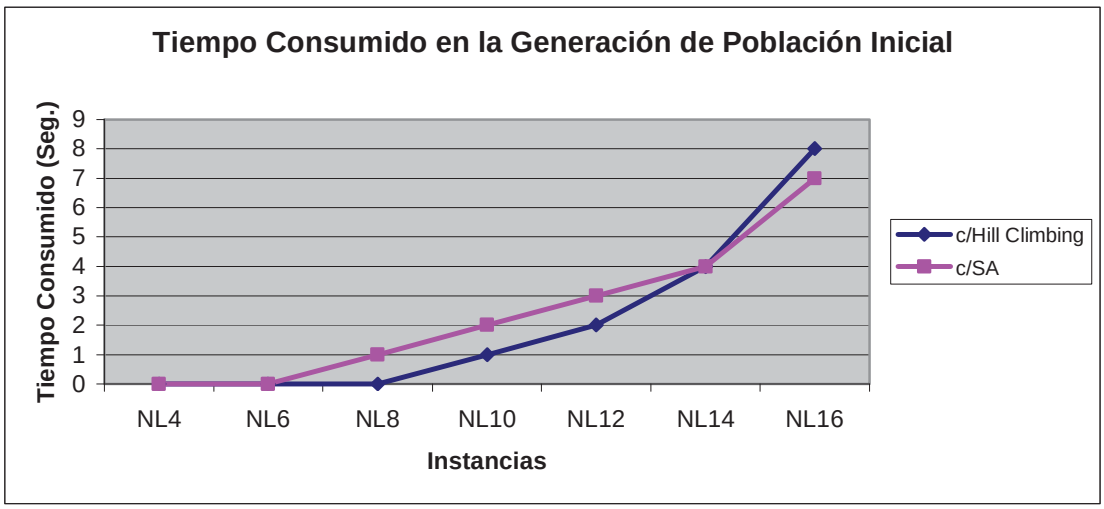


Figura 6.4.- Tiempo en Generación de Población

CONCLUSIONES

Con respecto a la calidad, puede verse que el optimizador local basado en Simulated Annealing obtiene una calidad superior en la generación de soluciones. En las instancias más pequeñas esto no es tan importante puesto que el espacio de búsqueda aunque es muy grande, puede tratarse eficientemente con ambas técnicas. Pero en las instancias grandes donde cada mejora requiere un esfuerzo mayor, se valora enormemente el aporte de SA ya que arroja

diferenta de hasta 6000 kilómetros de ahorro, lo que para esa instancia puede parecer pequeño, pero en realidad es un gran ahorro si lo lleváramos a dinero.

Por esta razón particularmente en esta etapa se concluye que el componente SA debe sustituir al Hill Climbing original para partir con una población de mejor calidad, esto nos ahorrará tiempo de cómputo en el ciclo evolutivo ya que el criterio de parada representan número de ciclos sin mejora, puede reducirse el número de mejoras en la evolución y por ende en número de ciclos.

Los tiempos analizados reflejan que no existe una diferencia sustancial para tomar una decisión contraria a la antes planteada.

6.2.2.- Pruebas De Movimientos

Esta prueba trata de demostrar que movimiento definido es el más beneficioso, con el fin de adoptar alguna estrategia de aplicación de dichos movimientos.

La estrategia de Prueba será la de crear la población inicial sin búsqueda local y luego aplicar a cada agente uno de los movimientos. Luego de haber probado todos, se podrá concluir cual es el que logra mejoras más significativas en la población.

La prueba fijará los parámetros para aislar el comportamiento del algoritmo en cuanto a la variación de las transformaciones y su calidad para encontrar mejoras. La cantidad de agentes será 20, y la instancia a tratar será NL10.

Los movimientos que se probarán serán los definidos y se medirán en comparación a los otros y a la generación de la población sin la aplicación de movimiento alguno. La *Tabla 6.6* muestra los fitness obtenidos para los 20 agentes creados, aplicando los diferentes movimientos. La *Figura 6.5* muestra esta situación gráficamente.

a)

	MOVIMIENTOS DEFINIDOS			
Agentes	***IPR***	***IR***	***IL***	***S/Movimiento***
1	85541	76229	79617	87841
2	85976	80010	81695	88139
3	87187	83447	81793	88763
4	87718	84029	82166	88933
5	89267	84096	82868	89429
6	89658	84713	83060	89467
7	89669	85562	83327	90006
8	90204	85820	83436	90028
9	90486	86380	83854	90327
10	90675	89722	83903	90966
11	91149	90166	84403	91497
12	91376	90456	84498	91839
13	91462	92502	84509	92204
14	91875	92797	85174	92444
15	92011	92840	85353	92445
16	92150	92861	85461	93368
17	92322	93047	85855	94121
18	92589	93357	87063	94159
19	94415	93874	88149	94498
20	94734	94868	89897	96122

Tabla 6.6.- Calidad en los Movimientos

b)

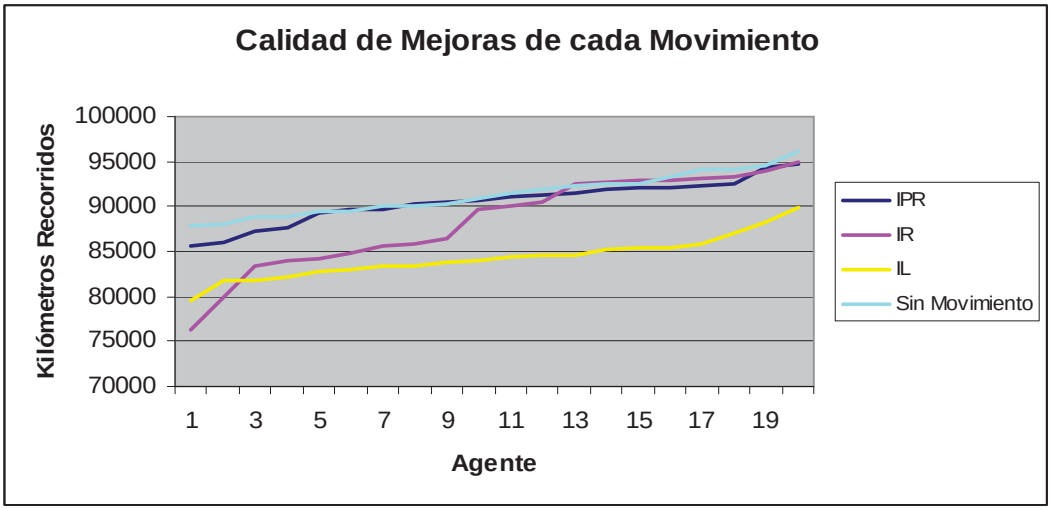


Figura 6.5.- Calidad en los Movimientos

CONCLUSIONES

Se puede ver que el mejor movimiento sin duda es el Intercambio de Localía (IL), ya que logra mejorar la población en su conjunto y arroja resultados dentro de un rango superior que los demás movimientos. El intercambio de rondas está en segundo lugar y el Intercambio Parcial de Rondas en tercero. Cabe mencionar que si bien el IPR arroja calidades peores que las demás, también es cierto que se aplica mucho menos veces en un Fixture que los otros dos métodos. Por esta razón, podemos decir que una aplicación de IPR es superior a una aplicación de los otros movimientos en promedio.

Por lo anterior se llamará a la función IPR una cantidad de veces mayor que a los demás movimientos. Aunque la clave claramente está en el movimiento de intercambio de localías.

6.2.3.- Pruebas En La Selección

Las pruebas en esta etapa pretenden comprobar la idea del beneficio de la diversidad, y que tan favorable para esta es el método adoptado (Torneo). Esto se realizará mediante la variación del parámetro “cardinalidad”, el cual representa el número de agentes que formarán el subgrupo desde donde se elegirán los padres de los próximos hijos. Teóricamente un número grande no es favorable por la perdida de diversidad, convergiendo rápido a un óptimo local. Mientras que un número pequeño produce que la evolución sea demasiado lenta por la probabilidad alta de elegir padres de calidad baja [17].

a) La prueba comienza probando el algoritmo completo para una instancia, una cantidad fija de agentes y un criterio fijo de parada para las diferentes ejecuciones.

Cantidad de Agentes = 20; Criterio de Parada = 1000 ciclos sin mejora; Instancia = NL14.

Algoritmo Completo NL14	
Cardinalidad	***Resultado***
2	264561
9	270404
16	263764
22	268518
30	271279

Tabla 6.7.- Pruebas de Cardinalidad

b) Se comprueba que para cardinalidades más grandes, se eligen mejores padres puesto que el subgrupo abarca mas agentes de la población.

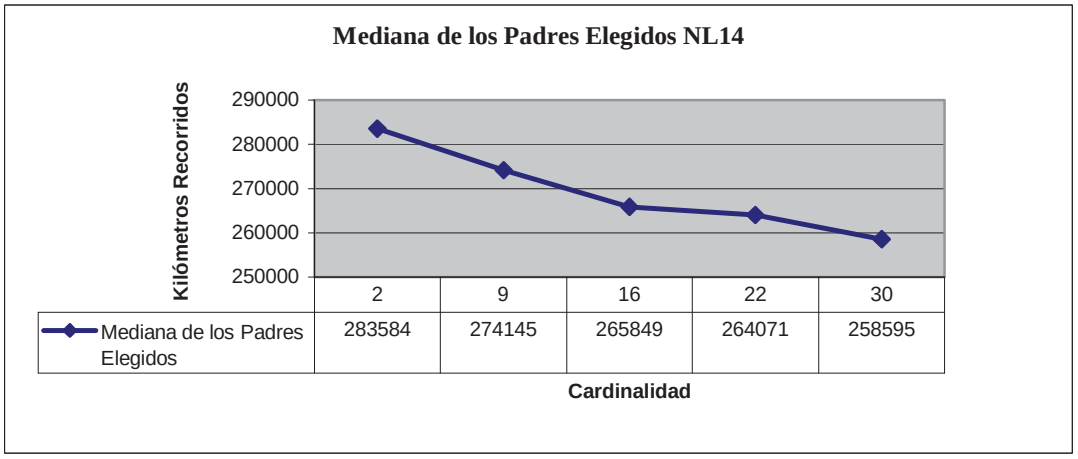


Figura 6.6.- Gráfico Calidad de los padres ante aumento de Cardinalidad.

CONCLUSIONES

Como se esperaba, el comportamiento del método de selección implementado va de acuerdo a la teoría, puesto que para cardinalidades grandes el algoritmo no arrojó mejores resultados, y para cardinalidades muy pequeñas los resultados no superaban a los obtenidos con cardinalidades intermedias. Esto comprueba que la diversidad es importante, pero en abundancia hace que el algoritmo no alcance a evolucionar a las soluciones con la velocidad requerida. Por lo tanto una cardinalidad alrededor de la media en la cantidad de agentes es una buena elección para las ejecuciones futuras.

Nota: Los resultados de la *Tabla 6.7* son medianas de 20 ejecuciones del algoritmo bajo las condiciones impuestas al principio. La mediana de los padres elegidos mostrados en la *Figura 6.6* se calculó de mil padres elegidos.

6.2.4.- Pruebas De Recombinación

Las pruebas de Recombinación tratan de comprobar que tan eficiente y eficaz es el operador creado para el problema para la creación de hijos. Las pruebas se basaron en dos aspectos, capacidad de mejora de los individuos y cantidad de recombinaciones hechas realmente ante una cantidad fija de ciclos.

Para esto se tomó el algoritmo y se eliminaron las dos fases de búsqueda local, así cualquier mejora sería resultado de la Recombinación. Los parámetros se fijaron, usando una cardinalidad igual a 12, cantidad de agentes igual a 20 y todas las instancias del problema, para diferentes cantidades de ciclos.

a.1)

Cantidad de Agentes = 20; # = 12	
Cantidad de Ciclos = 1000	
Instancia	***Resultado***
NL4	8413
NL6	27793
NL8	49258
NL10	83350
NL12	158070
NL14	301156
NL16	438225

Cantidad de Agentes = 20; # = 12	
Cantidad de Ciclos = 10000	
Instancia	***Resultado***
NL4	8392
NL6	27753
NL8	47662
NL10	82241
NL12	157285
NL14	300327
NL16	436304

Cantidad de Agentes = 20; # = 12	
Cantidad de Ciclos = 20000	
Instancia	***Resultado***
NL4	8276
NL6	28986
NL8	46906
NL10	80717
NL12	155730
NL14	298515
NL16	430871

Tabla 6.8.- Calidad de Soluciones Obtenidas por Recombinación

a.2)

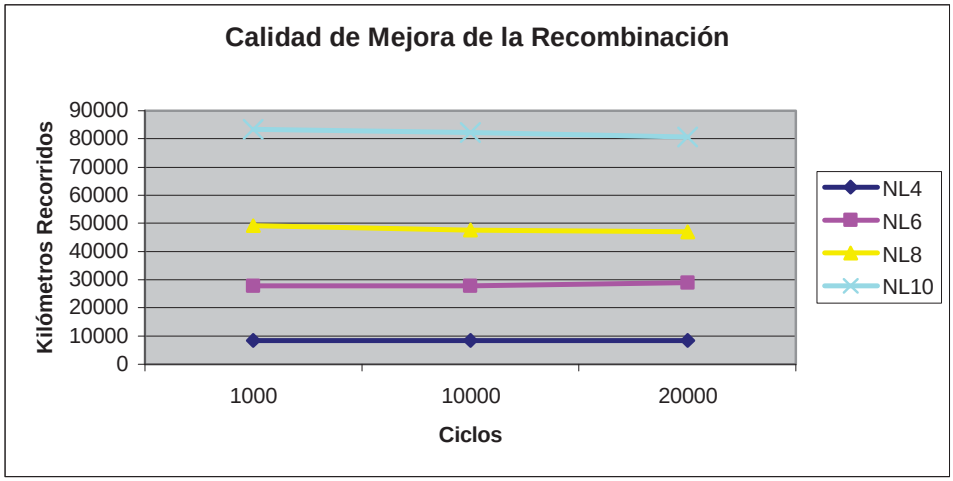


Figura 6.7.- Calidad de Resultados de la Recombinación (primera parte).

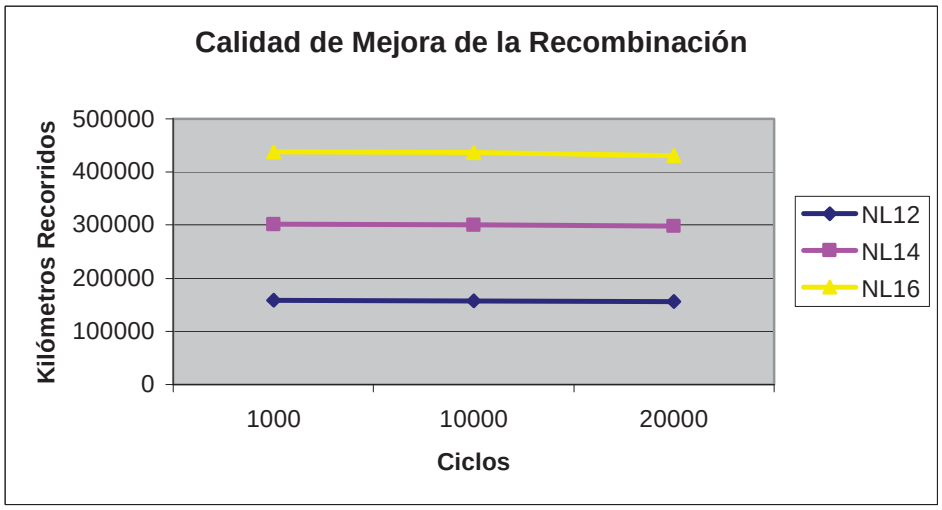


Figura 6.8.- Calidad de Resultados de la Recombinación (segunda parte).

b.1) Recombinaciones Realmente Hechas:

Para esta prueba se fijaron todos los parámetros y se contabilizaron las Recombinaciones que generaban hijos “sanos”, recordando que si el proceso de Recombinación no podía terminar de completar el fixture del hijo, se le asignaba un fitnnes muy alto para que al ingresar a la población muriese instantáneamente.

La *Tabla 6.9* muestra el porcentaje de hijos sanos que se generaron después de llamar mil veces al proceso de Recombinación.

Cantidad de Agentes = 20; # = 12	
Cantidad de Ciclos = 1000	
Instancia	***% Exito***
NL4	100
NL6	91,3
NL8	48,8
NL10	32,4
NL12	20,8
NL14	13,1
NL16	11,4

Tabla 6.9.- Eficiencia de Recombinación

b.2)

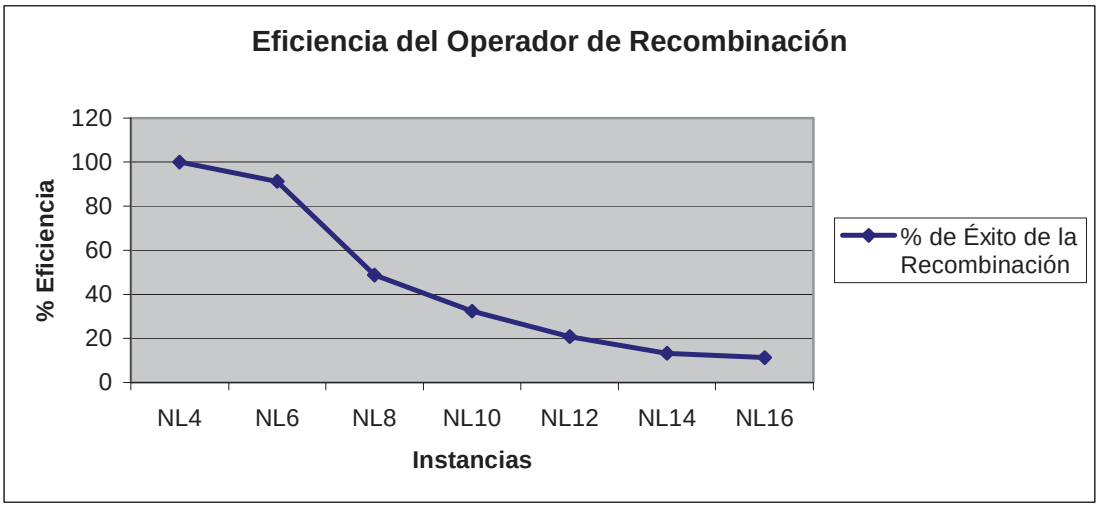


Figura 6.9.- Porcentaje de Éxito en la generación de hijos

CONCLUSIONES

La *Tabla 6.8* demuestra que el operador de recombinación si bien mejora las soluciones iniciales, no es capaz de acercarse prontamente a un óptimo, se ve que tarda muchos ciclos en demostrar mejoras decentes en la calidad de las soluciones, lo que sumado a que el proceso completo no soportaría una cantidad de ciclos muy grande, el operador de recombinación no cumpliría un factor relevante en el proceso de optimización sino que ayudaría a la exploración del espacio, ya que la manera en que está construida simula varios movimientos en uno para una solución determinada.

Por otro lado, en la T*abla 6,9* se ve que la cantidad de hijos que genera la recombinación es aceptable hasta la instancia NL12, lo que nos deja en un problema para las instancias mayores ya que al no diversificar la búsqueda es difícil que el operador de búsqueda local logre mejoras. Para esto se barajan dos alternativas:

- Primero, aumentar la cantidad de veces que la recombinación manda a rellenar al hijo creado, esto aumentará la probabilidad de crear hijos sanos.
- Segundo, además de lo anterior realizar en la segunda fase de búsqueda local un llamado solamente al componente SA, puesto que tiene la habilidad de explorar muy bien cuando comienza su proceso, ya que la temperatura es alta al inicio.

6.3.- PRUEBAS GLOBALES

Estas pruebas tienen como objetivo principal el completar el estudio del algoritmo propuesto, tratando de lograr su mejor funcionamiento. Existen códigos alternativos que serán probados de forma de evaluar si son capaces de cooperar con la búsqueda o solo compiten con lo existente.

6.3.1.- Pruebas Del Criterio De Parada

Para realizar esto se realizan ejecuciones sucesivas de las diferentes instancias del problema para identificar el número de ciclo en que la búsqueda deja de ser de calidad. Se muestra el número de ciclos en que se encontró una mejora del mejor individuo en la población actual para las instancias relevantes.

INSTANCIA	***Nº DE CICLOS***			
NL8	50	148	1091	1140
NL10	11	90	413	
NL12	25	142	743	
NL14	126			

Tabla 6.10.- Nº de ciclo en que se encontró mejora

La *Tabla 6.10* muestra distintas ejecuciones del algoritmo, la cantidad de ciclos que se evaluaron fueron 2000, la población de 20 agentes y la cardinalidad igual a 12. Podemos apreciar que después de cierta cantidad de ciclos, la búsqueda se atrapa y no produce resultados. Este número varía para cada instancia, y los mostrados corresponden a medianas calculadas a 20 ejecuciones para cada instancia con las condiciones antes descritas. Lamentablemente los números no son aclaratorios y el número de ciclo donde baja el rendimiento de la búsqueda está dentro de rangos muy amplios, por lo que se tomará un valor cercano a uno de los mostrados, en este caso será 700.

Lo anterior quiere decir que si en la ejecución del algoritmo para cualquier instancia si se llega a un número de ciclos igual a 700, se cambiará la estrategia de búsqueda. Esto se realizará de manera de cambiar el optimizador local en el ciclo evolutivo, de Hill Climbing a Simulated Annealing.

Para finalizar la evaluación del criterio de parada es importante destacar que un número mayor en el criterio de parada aumenta la probabilidad de encontrar una mejora, pero después de un cierto valor este parámetro pierde importancia por lo visto anteriormente.

6.3.2.- Pruebas Del Tamaño De La Población

El objetivo de esta prueba es determinar la influencia del tamaño de la población en el desarrollo del algoritmo. Para esto se revisarán ejecuciones de la instancia NL8, fijando los demás parámetros como la cardinalidad y el criterio de parada.

PARAMETROS	***EJECUCIONES***		
Tamaño de la Población	20	25	30
Cardinalidad	10	10	10
Criterio de Parada	1000	1000	1000
Resultado	43376	43172	42589

Tabla 6.11.- Mejoras con aumento de la población NL8

La *Tabla 6.11* muestra la mejora para los resultados en la instancia NL8 con el aumento de la población. Para instancias más grandes es necesario utilizar un tamaño de población cercano a 60 para aumentar la probabilidad de explorar espacios donde se encuentren óptimos. Para instancias pequeñas no se justifica utilizar tamaños mayores a 15, es más, para el NL4 basta con dos agentes como se verá en la *Tabla 6.12.*

PARAMETROS	***EJECUCIONES***		
Tamaño de la Población	4	2	**2**
Cardinalidad	2	2	**2**
Criterio de Parada	500	200	**50**
Resultado	8276	8276	**8276**

Tabla 6.12.- Mejoras con el aumento de la Población NL4.

6.3.3.- Pruebas Con Códigos Alternativos

Se probará el algoritmo mezclando el código alternativo realizado (Simulated Annealing) de manera de buscar el mejor aprovechamiento de este. Se pretende clarificar si los optimizadotes locales pueden cooperar o solo compiten.

Se realizarán pruebas para todas las instancias bajo las siguientes condiciones:

- Tamaño de la Población : 60 Agentes.
- Cardinalidad : 45.
- Criterio de Parada : 4000 ciclos sin mejora.

Con lo anterior se probará el algoritmo con las siguientes configuraciones:

- *AM-HC-HC:* Algoritmo Memético con la primera y segunda fase de Optimización Local aplicando Hill Climbing (H-C).
- *AM-SA-SA*: Algoritmo Memético con la primera y segunda fase de Optimización Local aplicando Simulated Annealing (SA).
- *AM-HC-SA*: Algoritmo Memético con la primera fase de búsqueda local aplicando H-C y la segunda SA.
- *AM-SA-HC*: Algoritmo Memético con la primera fase de búsqueda local aplicando SA y la segunda HC.

a) *AM-HC-HC y AM-SA-SA*

Instancia	*Resultado*	*Tiempo(Seg.)*
NL4	8276	1
NL6	26588	1
NL8	42479	35
NL10	73146	55
NL12	140272	112
NL14	262045	155
NL16	362102	302

Tabla 6.13.- AM-HC-HC

Instancia	*Resultado*	*Tiempo(Seg.)*
NL4	8276	2
NL6	26588	2
NL8	42654	43
NL10	72301	62
NL12	140059	110
NL14	249465	163
NL16	358991	334

Tabla 6.14.- AM-SA-SA

Las *Tablas 6.13* y *6.14* nos indican que SA es más eficiente para espacios de búsqueda más grandes. Esto sucede por que diversifica la búsqueda mejor que Hill Climbing que solo es un escalador [23].

b) *AM-HC.SA y AM-SA-HC*

Instancia	*Resultado*	*Tiempo(Seg.)*
NL4	8276	10
NL6	26588	15
NL8	42266	64
NL10	72027	54
NL12	135459	77
NL14	255879	185
NL16	355876	583

Tabla 6.15.- AM-SA-HC

Instancia	*Resultado*	*Tiempo(Seg.)*
NL4	8276	15
NL6	26588	22
NL8	43213	131
NL10	70650	146
NL12	136966	149
NL14	266828	271
NL16	362145	524

Tabla 6.16.- AM-HC-SA

Analizando las *Tablas 6.15* y 6.*16* puede verse que la alternativa *AM-SA-HC* tiene mejor rendimiento, y viene a comprobar que SA como primer optimizador tiene mejores resultados que HC (*Tabla 6.2, 6.3, 6.4*), esto trae como consecuencia que los padres sean de mejor calidad y la evolución se produzca más rápido.

c) *Cooperación SA-HC-SA*

En la *Sección 6.3.1* se dijo que la evolución pierde rendimiento avanzado un cierto número de ciclos. Además se definió un valor igual a 700 en el número de ciclos sin mejora para cambiar la estrategia de búsqueda (optimizador local). Originalmente se tendrá un operador de Hill Climbing trabajando dentro del ciclo evolutivo, pasado 700 ciclos sin mejoras, se cambiará a Simulated Annealing para abrir el espacio de búsqueda con una temperatura inicial alta.

La *Tabla 6.17* muestra resultados, algunos mejoran los obtenidos hasta el momento:

Instancia	***Resultado***	***Tiempo(Seg.)***
NL4	8276	-
NL6	26588	-
NL8	42218	204
NL10	69774	553
NL12	132239	935
NL14	253146	1116
NL16	351530	3003

Tabla 6.17.- Resultados Cooperación SA-HC

CONCLUSIONES

El presente informe entregó un resumen del marco teórico de los Algoritmos Meméticos como metaheurísticas evolutivas y del problema de la optimización de fixtures deportivos fundamentado en un problema formal llamado *Traveling Tournament Problem*, específicamente la instancia espejada.

Se realizó una segunda etapa donde se diseñó la solución del problema con base en la técnica y se definieron las etapas que decía cumplir el algoritmo. Finalmente se implementó el algoritmo en lenguaje C y se realizaron pruebas con alternativas de código.

De la implementación del algoritmo se pueden sacar conclusiones que ayudan a explicar la forma en que se comporta el algoritmo, algunas son:

a) La generación de la población inicial se realiza eficientemente por el método propuesto. La calidad de las soluciones resultantes de este método no es del todo buena, lo que es remediado por la primera fase de optimización local. Se estableció que utilizar Simulated Annealing en esta fase tiene un mejor rendimiento y el aumento en el tiempo no es significativo, además el partir el proceso evolutivo con buenas soluciones asegura una evolución más rápida. El tamaño de la población es un parámetro relacionado y se probó que una cantidad mayor de agentes beneficia la búsqueda por que diversifica el proceso al tener acceso a diferentes puntos en el espacio.

b) Dentro del operador de búsqueda local, existen transformaciones que son más beneficiosas que otras en cuanto a la mejora que producen a las soluciones y el ritmo en que lo hacen. Es así como el Intercambio de Localías demostró ser el más constante método de mejora de soluciones.

c) El método de selección tiene la importancia de influir en la evolución al elegir los padres que se reproducirán en hijos de calidad semejante o superior. La cardinalidad es el parámetro relacionado, y su influencia se manifiesta en el cambio en sus valores, si se

aproxima al tamaño de la población se pierde diversidad y la búsqueda se atrapa en óptimos locales. Por otro lado si el valor es muy pequeño, la evolución se hace lenta por la alta probabilidad de elegir padres de baja calidad.

d) El operador de Recombinación, tiene la capacidad de mejorar las soluciones por si misma pero su ritmo es demasiado lento. La efectividad de la Recombinación decrece con el aumento en la instancia ya que se hace cada ve más difícil completar fixtures de más equipos. La Recombinación también cumple un rol exploratorio ya que es capaz de generar fixtures que no están al alcance con una sola transformación, esto hace que el espacio de búsqueda cambie.

e) La cooperación en la segunda fase de búsqueda local entre HC y SA es la mejor alternativa ya que cuando HC se atrapa en un óptimo local, se puede salir de una manera diferente a la Recombinación con la aplicación de SA. Esto provoca que al comenzar, el proceso tenga una temperatura alta y pueda aceptar soluciones peores a las actuales y explorar otros espacios.

Finalmente el comportamiento multi-arranque de los AM favorece la exploración, y la combinación con métodos como el Hill Climbing y el Simulated Annealing potencian la explotación logrando buenos resultados para el problema tratado.

Sobre el MTTP se puede decir que aún está pendiente el encontrar valores mejores para las instancias grandes del problema. Además una vez comprendido se puede ampliar el área de trabajo, aplicando restricciones de un caso real para poder diseñar un software que optimice Fixtures desde el punto de vista no solo de los kilómetros recorridos, sino además de costos de viaje asociado a fechas en las cuales deben jugarse los partidos.

De manera especial quisiera mencionar que el trabajo presentado pone fin a muchos años de estudio, el cual enriqueció una parte importante de mi vida. Durante este tiempo más que números y teorías, aprendí que la vida no es más que una serie de consecuencias, y que las causas se construyen a diario como las líneas de un ferrocarril que no termina de viajar.

REFERENCIAS

[1] Gilberto Gil Junior, *"Algoritmos Meméticos aplicados ao problema de No-Wait Flowshop"*, Presentación de Maestría, Universidade Estadual de Campinas, 2001.

[2] Belén Melián, José A. Moreno Pérez, J. Marcos Moreno Vega, *"Metaheuristics: A Global View"*, Revista Iberoamericana de Inteligencia Artificial, (19): 7-28, 2003.

[3] Diana Holstein, *"Una Metaheuristica Co-evolutiva para el problema del viajante de Comercio"*, Tesis de Grado, Universidad Nacional de la Plata, 1998.

[4] Leandro dos Santos Coelho, *"Fundamentos, Potencialidades e Aplicacioes de Algoritmos Evolutivos"*, Editorial Sociedade Brasileira de Matemática Aplicada e Computacional (SBMAC) / Livraria Espaço, 2003.

[5] Hernán J. Dopazo, Roberto P. J. Perazzo, *"Aprendizaje y Evolución: Adaptación acelerada por Efecto Baldwin"*, J.A Hernández and A. Pomi (Eds.), 165-178, 1999.

[6] Gan Tiaw Leong, *"Constraint Programming for the Traveling Tournament Problem"*, Reporte de Proyecto, Universidad Nacional de Singapur, 2003.

[7] Martin Henz, Tobías Muller, Sven Thiel, *"Global Constraint for Round Robin Tournament Scheduling"*, European Journal for Operational Research, 92-101, 2002.

[8] M. A. Trick, K. Easton, G. Nemhauser, *"Solving the Traveling Tournament Problem: A Combined Integer Programming and Constraint Programming Approach"*, Proceedings of the 4th International Conference PATAT, 319-331, 2002.

[9] Pablo Moscato, *"On Evolution, Search Optimization, Genetic Algorithms and martial Arts: Towards memetic Algorithms"*, Caltech Programa de Computación Concurrente, C3P Reporte 826, 1989.

[10] Pablo Moscato, Carlos Cotta, *"An Introduction to Memetic Algorithms"*, Revista Iberoamericana de Inteligencia Artificial , (19):131-148, 2003.

[11] Fred Glover, Gary A. Kochenberger, *"Handbook of Metaheuristics"*, Kluver Academic Publisher, 2003.

[12] Nicholas J. Radcliffe, Patrick D. Surry, *"Formal Memetic Algorithms"*, Evolutionary Computing, AISB Workshop: 1-16, 1994.

[13] M. A. Trick, G. Nemhauser, *"Scheduling a Major College Basketball Conference"*, Operations Research**,** 46(1): 324–343, 1998.

[14] M. A. Trick, K. Easton, G. Nemhauser, *"The Traveling Tournament Problem, description and benchmarks"*, en ediciones T.Walsh, *Principios y Prácticas de la Programación con Restricciones*, notas de ciencias de la computación, (2239):580-584, 2001.

[15] Andrés Cardemil, *"Optimización de Fixtures Deportivos: Estado del Arte y un Algoritmo Tabú Search para el Traveling Tournament Problem"*, Tesis de Licenciatura de la Universidad de Buenos Aires, Argentina, 2002.

[16] David Alejandro Pelta, *"Algoritmos Heurísticos en Bioinformática"*, Tesis Doctoral, Universidad de Granada, 2000.

[17] Pablo Estévez Valencia, *"Optimización mediante Algoritmos Genéticos"*, Anales del Instituto de Ingenieros de Chile: 109 (2): 83-92, 1997.

[18] N. Krasnogor, S. Gustafson, *"La Búsqueda Local como proveedora de Bloques Constructivos en Algoritmos Meméticos Autogenerados"*, paper, 2003.

[19] Alexadre Mendez, Luciana Buriol, Pablo Moscato, Paulo Franca, *"The Memepool Project: A Framework for Combinatorial Optimizacion"*, presentación, www.densis.fee.unicamp,br/MEMEPOOL.

[20] Pablo Moscato, *"On Genetic Crossover Operator for Relative Order Preservation"*, Caltech Programa de computación concurrente, Instituto de Tecnología de California, C3P Report 778, 1990.

[21] Celso C. Ribeiro, Sebastian Urrutia, *"Heuristics for the Mirrored Traveling Tournament Problem"*, PATAT 2004 - Practice and Theory of Automated Timetabling, 323-342, 2004.

[22] Ricardo Concilio, Fernando J. Von Zuben, *"Evolutionary Design of Schedules in Championships with Compact Genetic Codification and Local Search"*, Genetic Evolutionary Computation Conference, Estados Unidos, 1:109-113, 2000.

[23] Julio Brito S., Clara Campos R., Félix García L., Miguel García T., Belén Melián B., José moreno P., J. Moreno Vega, *"Metaheurísticas: Una revisión actualizada"*, Documentos de Trabajo del DEIOC. Universidad de La Laguna, Número 2/2004.

[24] A. Lim, B. Rodrigues, X. Zhang, *"A Simulated Annealing and Hill-Climbing Algorithm for the Traveling Tournament Problem"*, European Journal of Operational Research, paper por aparecer, 2005.

ANEXOS

ANEXO 1: EJECUCION DEL ALGORITMO PARA INSTANCIA NL4.

Mejor Solución Encontrada

Rondas/Equipos	***1***	***2***	***3***	***4***
1	3	-4	-1	2
2	2	-1	4	-3
3	4	-3	2	-1
4	-3	4	1	-2
5	-2	1	-4	3
6	-4	3	-2	1

Figura A.1.- Mejor Resultado NL4

Recorrido Total Hijo: 8276

Tiempo de Búsqueda: 1 segundos

ANEXO 2: EJECUCION DEL ALGORITMO PARA INSTANCIA NL6.

Mejor Resultado

Rondas/Equipos	***1***	***2***	***3***	***4***	***5***	***6***
1	4	-3	2	-1	-6	5
2	2	-1	-6	-5	4	3
3	3	-5	-1	6	2	-4
4	-6	4	-5	-2	3	1
5	5	-6	4	-3	-1	2
6	-4	3	-2	1	6	-5
7	-2	1	6	5	-4	-3
8	-3	5	1	-6	-2	4
9	6	-4	5	2	-3	-1
10	-5	6	-4	3	1	-2

Figura A.2.- Mejor Resultado NL6

Recorrido Total Hijo: 26588

Tiempo de Búsqueda: 2 segundos

ANEXO 3: EJECUCION DEL ALGORITMO PARA INSTANCIA NL8.

Mejor Resultado

Rondas/Equipos	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
1	7	-4	6	2	-8	-3	-1	5
2	-8	6	4	-3	-7	-2	5	1
3	-4	7	8	1	-6	5	-2	-3
4	-6	8	-5	7	3	1	-4	-2
5	3	-5	-1	8	2	-7	6	-4
6	2	-1	-7	-5	4	-8	3	6
7	5	-3	2	-6	-1	4	-8	7
8	-7	4	-6	-2	8	3	1	-5
9	8	-6	-4	3	7	2	-5	-1
10	4	-7	-8	-1	6	-5	2	3
11	6	-8	5	-7	-3	-1	4	2
12	-3	5	1	-8	-2	7	-6	4
13	-2	1	7	5	-4	8	-3	-6
14	-5	3	-2	6	1	-4	8	-7

Figura A.3.- Mejor Resultado NL8

Recorrido Total Hijo: 42218

Tiempo de Búsqueda: 204 segundos

ANEXO 4: EJECUCION DEL ALGORITMO PARA INSTANCIA NL10.

Mejor Resultado.

Recorrido Total Hijo: 69774. Tiempo de Búsqueda: 553 segundos

Rondas/Equipos	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
1	8	6	-5	-7	3	-2	4	-1	-10	9
2	10	-5	-7	-9	2	8	3	-6	4	-1
3	-3	-7	1	6	-9	-4	2	10	5	-8
4	-4	-10	6	1	-8	-3	-9	5	7	2
5	9	8	-4	3	-10	7	-6	-2	-1	5
6	7	9	8	-5	4	10	-1	-3	-2	-6
7	-2	1	9	-10	6	-5	8	-7	-3	4
8	-5	-3	2	-8	1	-9	-10	4	6	7
9	6	-4	-10	2	-7	-1	5	-9	8	3
10	-8	-6	5	7	-3	2	-4	1	10	-9
11	-10	5	7	9	-2	-8	-3	6	-4	1
12	3	7	-1	-6	9	4	-2	-10	-5	8
13	4	10	-6	-1	8	3	9	-5	-7	-2
14	-9	-8	4	-3	10	-7	6	2	1	-5
15	-7	-9	-8	5	-4	-10	1	3	2	6
16	2	-1	-9	10	-6	5	-8	7	3	-4
17	5	3	-2	8	-1	9	10	-4	-6	-7
18	-6	4	10	-2	7	1	-5	9	-8	-3

Figura A.4.- Mejor Resultado NL10

ANEXO 5: EJECUCION DEL ALGORITMO PARA INSTANCIA NL12.

Mejor Resultado

Rondas/Equipos	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
1	2	-1	-7	11	-10	9	3	-12	-6	5	-4	8
2	7	11	-10	9	-8	12	-1	5	-4	3	-2	-6
3	-11	9	8	12	6	-5	10	-3	-2	-7	1	-4
4	10	12	-6	-5	4	3	8	-7	11	-1	-9	-2
5	-9	-5	4	-3	2	7	-6	-10	1	8	-12	11
6	-8	-3	2	7	-11	10	-4	1	-12	-6	5	9
7	-12	-7	-11	10	9	-8	2	6	-5	-4	3	1
8	6	10	9	8	12	-1	-11	-4	-3	-2	7	-5
9	-5	8	12	-6	1	4	-9	-2	7	11	-10	-3
10	4	-6	5	-1	-3	2	12	11	-10	9	-8	-7
11	3	-4	-1	2	-7	11	5	9	-8	-12	-6	10
12	-2	1	7	-11	10	-9	-3	12	6	-5	4	-8
13	-7	-11	10	-9	8	-12	1	-5	4	-3	2	6
14	11	-9	-8	-12	-6	5	-10	3	2	7	-1	4
15	-10	-12	6	5	-4	-3	-8	7	-11	1	9	2
16	9	5	-4	3	-2	-7	6	10	-1	-8	12	-11
17	8	3	-2	-7	11	-10	4	-1	12	6	-5	-9
18	12	7	11	-10	-9	8	-2	-6	5	4	-3	-1
19	-6	-10	-9	-8	-12	1	11	4	3	2	-7	5
20	5	-8	-12	6	-1	-4	9	2	-7	-11	10	3
21	-4	6	-5	1	3	-2	-12	-11	10	-9	8	7
22	-3	4	1	-2	7	-11	-5	-9	8	12	6	-10

Figura A.5.- Mejor Resultado NL12

Recorrido Total Hijo: 132239

Tiempo de Búsqueda : 935 segundos.

ANEXO 6: EJECUCION DEL ALGORITMO PARA INSTANCIA NL14.

Mejor Resultado

Rondas/Equipos	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***
1	3	-5	-1	14	2	12	13	-11	-10	9	8	-6	-7	-4
2	2	-1	13	-11	10	-9	-8	7	6	-5	4	14	-3	-12
3	-12	13	10	-9	8	-7	6	-5	4	-3	-14	1	-2	11
4	-13	-10	8	-7	-6	5	4	-3	-14	2	12	-11	1	9
5	11	8	-6	5	-4	3	-14	-2	-12	13	-1	9	-10	7
6	-10	6	-4	3	14	-2	-12	13	-11	1	9	7	-8	-5
7	-9	-4	14	2	12	13	-11	10	1	-8	7	-5	-6	-3
8	8	14	12	-13	11	10	9	-1	-7	-6	-5	-3	4	-2
9	7	12	11	-10	-9	8	-1	-6	5	4	-3	-2	-14	13
10	6	11	-9	-8	-7	-1	5	4	3	-14	-2	13	-12	10
11	-5	-9	-7	6	1	-4	3	14	2	-12	13	10	-11	-8
12	4	-7	-5	-1	3	14	2	12	-13	-11	10	-8	9	-6
13	14	-3	2	-12	13	11	-10	-9	8	7	-6	4	-5	-1
14	-3	5	1	-14	-2	-12	-13	11	10	-9	-8	6	7	4
15	-2	1	-13	11	-10	9	8	-7	-6	5	-4	-14	3	12
16	12	-13	-10	9	-8	7	-6	5	-4	3	14	-1	2	-11
17	13	10	-8	7	6	-5	-4	3	14	-2	-12	11	-1	-9
18	-11	-8	6	-5	4	-3	14	2	12	-13	1	-9	10	-7
19	10	-6	4	-3	-14	2	12	-13	11	-1	-9	-7	8	5
20	9	4	-14	-2	-12	-13	11	-10	-1	8	-7	5	6	3
21	-8	-14	-12	13	-11	-10	-9	1	7	6	5	3	-4	2
22	-7	-12	-11	10	9	-8	1	6	-5	-4	3	2	14	-13
23	-6	-11	9	8	7	1	-5	-4	-3	14	2	-13	12	-10
24	5	9	7	-6	-1	4	-3	-14	-2	12	-13	-10	11	8
25	-4	7	5	1	-3	-14	-2	-12	13	11	-10	8	-9	6
26	-14	3	-2	12	-13	-11	10	9	-8	-7	6	-4	5	1

Figura A.6.- Mejor Resultado NL14

Recorrido Total Hijo: 253146

Tiempo de Búsqueda: 1116 segundos.

ANEXO 7: EJECUCION DEL ALGORITMO PARA INSTANCIA NL16.

Mejor Resultado

Ronda/Equipos	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
1	-16	-3	2	-15	14	13	12	11	-10	9	-8	-7	-6	-5	4	1
2	-2	1	-14	-13	12	11	-10	9	-8	7	-6	-5	4	3	16	-15
3	15	-14	-12	-11	10	-9	8	-7	6	-5	4	3	16	2	-1	-13
4	-14	-12	10	9	-8	-7	6	5	-4	-3	-16	2	15	1	-13	11
5	13	10	8	-7	6	-5	4	-3	16	-2	-15	14	-1	-12	11	-9
6	12	8	-6	-5	4	3	-16	-2	15	14	-13	-1	11	-10	-9	7
7	-11	-6	4	-3	-16	2	15	14	13	12	1	-10	-9	-8	-7	5
8	10	-4	-16	2	15	-14	13	12	-11	-1	9	-8	-7	6	-5	3
9	9	16	-15	-14	13	-12	-11	-10	-1	8	7	6	-5	4	3	-2
10	-8	-15	-13	-12	11	10	-9	1	7	-6	-5	4	3	-16	2	14
11	7	-13	11	10	-9	8	-1	-6	5	-4	-3	-16	2	15	-14	12
12	6	11	9	8	-7	-1	5	-4	-3	16	-2	-15	-14	13	12	-10
13	-5	9	-7	-6	1	4	3	16	-2	15	14	13	-12	-11	-10	-8
14	4	-7	-5	-1	3	16	2	15	14	-13	12	-11	10	-9	-8	-6
15	-3	-5	1	-16	2	-15	-14	-13	-12	11	-10	9	8	7	6	4
16	16	3	-2	15	-14	-13	-12	-11	10	-9	8	7	6	5	-4	-1
17	2	-1	14	13	-12	-11	10	-9	8	-7	6	5	-4	-3	-16	15
18	-15	14	12	11	-10	9	-8	7	-6	5	-4	-3	-16	-2	1	13
19	14	12	-10	-9	8	7	-6	-5	4	3	16	-2	-15	-1	13	-11
20	-13	-10	-8	7	-6	5	-4	3	-16	2	15	-14	1	12	-11	9
21	-12	-8	6	5	-4	-3	16	2	-15	-14	13	1	-11	10	9	-7
22	11	6	-4	3	16	-2	-15	-14	-13	-12	-1	10	9	8	7	-5
23	-10	4	16	-2	-15	14	-13	-12	11	1	-9	8	7	-6	5	-3
24	-9	-16	15	14	-13	12	11	10	1	-8	-7	-6	5	-4	-3	2
25	8	15	13	12	-11	-10	9	-1	-7	6	5	-4	-3	16	-2	-14
26	-7	13	-11	-10	9	-8	1	6	-5	4	3	16	-2	-15	14	-12
27	-6	-11	-9	-8	7	1	-5	4	3	-16	2	15	14	-13	-12	10
28	5	-9	7	6	-1	-4	-3	-16	2	-15	-14	-13	12	11	10	8
29	-4	7	5	1	-3	-16	-2	-15	-14	13	-12	11	-10	9	8	6
30	3	5	-1	16	-2	15	14	13	12	-11	10	-9	-8	-7	-6	-4

Figura A.7.- Mejor Resultado NL16

Recorrido Total Hijo: 351530

Tiempo de Búsqueda: 3003 segundos.

ANEXO 8: CODIGO FUENTE.

A.8.1.- Cabecera.h

```
/* Archivos de cabecera validos para todos los demás*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <math.h>

/* Archivos de definición de funciones y procedimientos a utilizar*/

void escribir_matriz(int equipos, int**m);
void asignar_rival(int equipos, int** m, int *poligono);
int* crear_poligono(int equipos);
void iniciar_poligono(int equipos, int *poligono);
void liberar_poligono(int *poligono);
int** crear_fixture(int equipos);
void liberar_fixture(int equipos, int **m);
void asignar_localias(int equipos, int **m);
int calcular_n(int fila, int columna,int **m);
void llenar_cuadro(int equipos, int** distancia);
int** crear_cuadro(int equipos);
int evaluar_fixture(int equipos, int** distancia,int **m);
int evaluar_factibilidad(int equipos, int **m);
void ir(int equipos,int **m, int **distancia);
void ir_2(int equipos,int **m, int **distancia);
void il(int equipos,int **m, int **distancia);
void il_hill(int equipos,int **m, int **distancia);
void ipr(int equipos,int **m,int **distancia);
void itl(int equipos,int **m, int **distancia);
void asignar_mejor_fixture(int **m, int **fixture_aux,int equipos);
void corregir_fixture(int equipos,int **m);
void eliminar_localias(int equipos,int **m);
void s_a(int equipos,int **m, int **distancia);
struct agente * crear_agente(int equipos,int **distancia);
struct padres * torneo(struct agente *pprimero,int cantidad,int cardinalidad);
struct agente * recombinacion(struct agente *padre1,struct agente *padre2, int equipos, int
**distancia);
void borrar_filas(int **m,int fila, int equipos);
int completa_hijo(int **hijo,int equipos);
int revisa_fila(int **hijo,int rival,int fila,int equipos);
int revisa_columna(int **hijo,int rival,int columna,int equipos);
```

```
struct agente * agregar_agente(struct agente *pprimero,struct agente *pnuevo);
void factibilizar_solucion(int **m, int equipos);
int dos_elevado(int equipos);
int signo_celda(int numero);
struct agente * elimina_ultimo(struct agente * pprimero);
double sim_ann(int cost_ant, int cost_act, double temperatura);

/* Definición de estructuras de datos necesarias y archivos*/

struct agente{
        struct agente *siguiente;
        int **matriz;
        int fitness;
};

struct padres{
        struct agente *padre1;
        struct agente *padre2;
};
FILE *resultados;
```

A.8.2.- Algoritmo Memético (memetico.c)

```
/* incluye los archivos de cabecera*/
#include "cabecera.h"

/* Principal */
main(){
int i,equipos,cantidad, criterio=0,cardinalidad=0;
int **distancia;
struct agente *pprimero,*pnuevo,*aux,*hijo,*pultimo;
struct padres *selec_padres;
int contador=0;
time_t t_inicio,t_poblacion,t_inicio_iteracion,t_termino;
srand ( time(NULL) );

pr_pob=fopen("./resultados/pob_ini.txt","a");
printf("Algoritmo Memetico aplicado al Mirrored Traveling Tournament Problem\n");
printf("              Por : John Barril A.    Proyecto 2.\n\n");

/* Ingreso de parámetros*/
printf("Ingrese el numero de equipos..:");scanf("%d",&equipos);
printf("\n\n");
printf("Ingrese cantidad de Agentes..:");scanf("%d",&cantidad);
printf("\n");
```

```
 printf("\n\nFase de seleccion\n\n Cardinalidad del Grupo:");
 do{
 cardinalidad=0;
 scanf("%d",&cardinalidad);
 }while(cardinalidad<2 || cardinalidad>cantidad);

/* Se rescatan las distancias desde el archivo que las contiene y se crea el cuadro*/
distancia=crear_cuadro(equipos);
llenar_cuadro(equipos,distancia);
pprimero=NULL;


(void) time(&t_inicio); /*comienza el tiempo*/

/****Creación de la Población Inicial*********/
for (i=0;i<cantidad;i++)
{
        pnuevo=crear_agente(equipos,distancia); /*el individuo pasa por la búsqueda local*/
        if (pprimero==NULL) pprimero=pnuevo;
        else pprimero=agregar_agente(pprimero,pnuevo);
  }

(void) time(&t_poblacion);

/* Se muestra la población inicial en pantalla*/
for (aux=pprimero;aux !=NULL;aux=(*aux).siguiente)
{  printf("\n");
   escribir_matriz(equipos,(*aux).matriz);
   printf("\nRecorrido Total:%d   %d\n",(*aux).fitness,aux);
}

/*Comienza el ciclo evolutivo*/

(void) time(&t_inicio_iteracion);
while (criterio<2500) /*criterio de parada 2500 ciclos sin mejora*/
{
   criterio++;
   aux = pprimero;
   selec_padres=torneo(pprimero,cantidad,cardinalidad); /* Proceso de selección de padres*/

  /* Los dos padres elegidos se pasan a Recombinación, recibiendo un hijo*/
   hijo=recombinacion((*selec_padres).padre1,(*selec_padres).padre2,equipos,distancia);

 /* El hijo pasa por la búsqueda local */
   hill_climbing( equipos, (*hijo).matriz, distancia );
```

```
/* Cooperación entre SA y Hill Climbing*/
 if (criterio > 700) s_a( equipos, (*hijo).matriz, distancia );
          else hill_climbing( equipos, (*hijo).matriz, distancia );

/* Si el hijo es defectuoso, se asigna un fitness muy alto*/
                    (*hijo).fitness=evaluar_fixture(equipos,distancia, (*hijo).matriz);
                    if ((*hijo).fitness==0)(*hijo).fitness=1000000;

/* Se agrega hijo a la población*/
                     pprimero=agregar_agente(pprimero,hijo);

/* Se verifica si existió mejora de la población*/
                      if (pprimero != aux)
                         {     criterio=0;
                               aux=pprimero;
                         }

                        pultimo= elimina_ultimo(pprimero);

                         contador=0;
                         while (aux != pultimo)
                         {     if ( (*aux).siguiente ==pultimo) break;
                               aux=(*aux).siguiente;
                               contador++;
                         }

/* Se elimina al últio agente de la población*/
                  liberar_fixture(equipos,(*pultimo).matriz);
                  free(pultimo);
                  (*aux).siguiente=NULL;
                  free(selec_padres);

} /* Loop ciclo evolutivo*/

 (void) time(&t_termino); /* toma el tiempo de termino*/

printf("\n\nMejor Solución Encontrada\n\n");
escribir_matriz(equipos,(*pprimero).matriz);
fprintf(pr_pob,"%d  \n",(*pprimero).fitness);
printf("\nRecorrido Total Hijo :%d  %d\n",(*pprimero).fitness,aux);
printf("\nTiempo de Busqueda : %d segundos", (int) t_termino-t_inicio);
}
```

A.8.3.- Creación de los Fixtures (Fixtures.c)

```
#include "cabecera.h"

/*Función que crea la matriz Mr,e en memoria*/
int** crear_fixture(int equipos)
{ int i,fil,col;
  int **m;
  m=malloc(sizeof(int*)* (2*(equipos-1)) );
        if (m==NULL) printf("No se ha podido pedir memoria para las filas");
                for(i=0;i<2*(equipos-1);i++)
                {
                        m[i]=malloc(sizeof(int*)* equipos);
                        if (m[i]==NULL) printf("para las columnas");
                }
                for (fil=0;fil<2*(equipos-1);fil++)
                {
                        for (col=0;col<equipos;col++)
                        {   m[fil][col]=0;
                        }
                }
return(m);
}

/*Procedimiento que asigna rivales mediante el polígono*/
 void asignar_rival(int equipos, int **m,int* poligono)
 {int fil,asc,desc,sw,i;

  for (fil=0;fil<equipos-1;fil++)
  {  desc=equipos-1;
          /*Asignación del equipo fuera del polígono*/
          m[fil][poligono[0]-1]=poligono[1];
          m[fil][poligono[1]-1]=poligono[0];
          m[fil+equipos-1][poligono[0]-1]=poligono[1];
          m[fil+equipos-1][poligono[1]-1]=poligono[0];

         /*Asignación poligonal*/
         for(asc=2;asc<(equipos/2)+1;asc++)
           {
                m[fil][poligono[asc]-1]=poligono[desc];
                m[fil][poligono[desc]-1]=poligono[asc];
                m[fil+equipos-1][poligono[asc]-1]=poligono[desc];
                m[fil+equipos-1][poligono[desc]-1]=poligono[asc];
                desc--;
           }
```

```
        /* Gira el polígono*/
        sw=poligono[equipos-1];
        for(i=equipos-1;i>1;i--)poligono[i]=poligono[i-1];
        poligono[1]=sw;
 }
}

/* Procedimiento de asignación de localías*/
 void asignar_localias(int equipos, int **m)
 {int nx,ny,i,j,estado;
  int local[2]={1,-1}; /* posibles estados*/

 /* Asignación aleatoria de la ronda 0 y (equipos-1) */
  for (i=0;i<equipos;i++)
   {estado=local[rand()%2];
          if (m[0][m[0][i]-1]>0 && m[0][i]>0)
          { m[0][m[0][i]-1]*=estado;
            m[0][i]*=-estado;
                m[equipos-1][m[equipos-1][i]-1]*=-estado;
                m[equipos-1][i]*=estado;

          }
   }

  /*comienzo de la heuristica*/
   for (i=1;i<equipos-1;i++)
   { for (j=0;j<equipos;j++)
        {
        if (m[i][m[i][j]-1]>0 && m[i][j]>0)
        {       nx=calcular_n(i,j,m);
                ny=calcular_n(i,m[i][j]-1,m);

                        if (nx>ny)
                        {
                                if (m[i-1][j]>0)
                                { m[i][j]*=-1;
                                  m[equipos+i-1][m[i+equipos-1][j]-1]*=-1;

                                }
                                else
                                {  m[i][m[i][j]-1]*=-1;
                                   m[equipos+i-1][j]*=-1;

                                }
                        }
                        if (ny>nx)
                        {
                                if (m[i-1][m[i][j]-1]>0)
                                { m[i][m[i][j]-1]*=-1;
```

```
                                m[equipos+i-1][j]*=-1;

                            }
                            else
                            {       m[i][j]*=-1;
                                    m[equipos+i-1][m[i+equipos-1][j]-1]*=-1;
                            }
                        }
                        if (ny==nx)
                        {estado=local[rand()%2];
                         m[i][j]*=estado;
                         m[i][m[i][j]-1]*=-estado;
                         m[equipos+i-1][j]*=-estado;
                         m[equipos+i-1][m[i+equipos-1][j]-1]*=estado;
                        }
        }


        }

  }

/* Verificación de validez del Fixture en U=3, y correción si es necesario*/
  if (evaluar_factibilidad(equipos,m)==0) corregir_fixture(equipos,m);

 }

/* Calculo de Nx y Ny*/
int calcular_n(int fila, int columna, int **m)
{int i,contador=0, acumulador=0;
        i=fila;
        while (i-1 >= 0)
        {
                if (m[i-1][columna]<0 && acumulador >0) break;
                if (m[i-1][columna]>0 && acumulador < 0) break;
                acumulador+= m[i-1][columna];
                contador++;
                 i--;
         }
 return(contador);
}
```

A.8.4.- Creación de la Población Inicial (Población_inicial.c)

```
#include "cabecera.h"


struct agente * crear_agente(int equipos,int **distancia)
{ struct agente *pnuevo;
  int * poligono; /* Se declara el polígono*/

  poligono=crear_poligono(equipos); /* Se crea el polígono  con un orden aleatorio*/
  iniciar_poligono(equipos,poligono);

  /* Se crea un nodo para un agente*/
  pnuevo=malloc(sizeof(struct agente*));

        (*pnuevo).matriz=crear_fixture(equipos); /* se crea la  matriz*/
        asignar_rival(equipos,(*pnuevo).matriz,poligono); /* Se asigna Rival */
        factibilizar_solucion((*pnuevo).matriz,equipos);
        s_a(equipos,(*pnuevo).matriz,distancia); /* Se aplica optimizador SA*/

       /* Se calcula el fitness del fixture y se agrega a la estructura*/
        (*pnuevo).fitness=evaluar_fixture(equipos,distancia,(*pnuevo).matriz);
        (*pnuevo).siguiente=NULL;

        liberar_poligono(poligono); /* Se libera memoria*/
        return(pnuevo); /*Retorna el puntero al Nodo creado*/
}


/*Función para agregar en orden un agente a la población*/
struct agente * agregar_agente(struct agente *pprimero,struct agente *pnuevo)
{ struct agente *recorre;
  recorre=pprimero;

  if ((*pnuevo).fitness < (*pprimero).fitness)
        {  (*pnuevo).siguiente=pprimero;
           pprimero=pnuevo;
        }
 else
        {       do{
                   if ((*recorre).siguiente == NULL)  (*recorre).siguiente=pnuevo;
                   else
                   if  ((*recorre).fitness<=(*pnuevo).fitness  &&
                                        (*(*recorre).siguiente).fitness>=(*pnuevo).fitness)
                   {  (*pnuevo).siguiente=(*recorre).siguiente;
                     (*recorre).siguiente=pnuevo;
                   }
                   recorre=(*recorre).siguiente;
```

```
                }while(recorre != pnuevo);
        }

return(pprimero); /* retorna un puntero al primer agente*/

}


/* Función que asigna localías hasta que el fixture sea valido para U=3*/
void factibilizar_solucion(int **m, int equipos)
{ int repita=0;

        do{
                asignar_localias(equipos,m);
                if (evaluar_factibilidad(equipos,m)==0)
                        {       repita=1;
                                eliminar_localias(equipos,m);
                        }
                else repita=0;
        }while (repita ==1);

}


/*Función que retorna el último nodo*/
struct agente * elimina_ultimo(struct agente * pprimero)
{ struct agente *aux=NULL,*pultimo=NULL;
   for (aux=pprimero;aux !=NULL;aux=(*aux).siguiente)
  { pultimo=aux;
  }
return(pultimo);

}
```

A.8.5.- Definición de Movimientos para Hill Climbing (búsqueda_local.c)

```
/* Intercambio Parcial de Rondas */

void ipr(int equipos,int **m,int **distancia)
{  int i,j,x,y, rev, cost_ant,cost_act;
   int **fixture_cambio;

        fixture_cambio=crear_fixture(equipos); /* Creación del fixture auxiliar */
        asignar_mejor_fixture(fixture_cambio,m,equipos); /* Se asigna m a fixture auxiliar*/
        rev=equipos-1;
  /* se recorre  m de manera normal */
    for (i=0;i<equipos-1;i++)
        for (j=0;j<equipos;j++)
                { /* se recorren las filas sucesivas para encontrar las condiciones del cambio*/

                 for (y=i+1;y<equipos-1;y++)
                 {   for(x=j+1;x<equipos;x++)
                        if (abs(m[i][x])==abs(m[y][j]) && abs(m[i][j])==abs(m[y][x]))
                            { /*Fixture auxiliar recibe los cambios */
                              fixture_cambio[i][j]= m[y][j];
                              fixture_cambio[i][abs(m[i][j])-1]= m[y][abs(m[i][j])-1];
                              fixture_cambio[i][x]= m[y][x];
                              fixture_cambio[i][abs(m[i][x])-1]= m[y][abs(m[i][x])-1];

                              fixture_cambio[y][j]= m[i][j];
                              fixture_cambio[y][abs(m[i][j])-1]= m[i][abs(m[i][j])-1];
                              fixture_cambio[y][x]=m[i][x];
                              fixture_cambio[y][abs(m[i][x])-1]= m[i][abs(m[i][x])-1];

                              fixture_cambio[i+rev][j]= m[y+rev][j];
                              fixture_cambio[i+rev][abs(m[i+rev][j])-1]=
m[y+rev][abs(m[i+rev][j])-1];
                              fixture_cambio[i+rev][x]= m[y+rev][x];
                              fixture_cambio[i+rev][abs(m[i+rev][x])-
1]=m[y+rev][abs(m[i+rev][x])-1];

                              fixture_cambio[y+rev][j]= m[i+rev][j];
                              fixture_cambio[y+rev][abs(m[i+rev][j])-1]=
m[i+rev][abs(m[i+rev][j])-1];
                              fixture_cambio[y+rev][x]= m[i+rev][x];
                              fixture_cambio[y+rev][abs(m[i+rev][x])-1]=
m[i+rev][abs(m[i+rev][x])-1];
                              cost_ant=evaluar_fixture(equipos,distancia,m);
                              cost_act=evaluar_fixture(equipos,distancia,fixture_cambio);
                              /* se evalúan los costos anterior y actual*/
                              if (cost_ant>cost_act)
                              {
                                 asignar_mejor_fixture(m,fixture_cambio,equipos);
```

```
                                    }
                                        else asignar_mejor_fixture(fixture_cambio,m,equipos);

                                }
                        }
                      }
    liberar_fixture(equipos,fixture_cambio); /* se libera el fixture auxiliary */

}


/* INTERCAMBIO DE RONDAS*/

void ir(int equipos,int **m, int **distancia)
{ int **fixture_aux;
  int *filas;
  int i,j,criterio,fila1,fila2;

  filas=crear_poligono(equipos-1); /* se crea un vector dinámico, aprovechando lo programado*/
  fixture_aux=crear_fixture(equipos); /* se crea un fixture auxiliar */

        for (i=0;i<equipos-1;i++) filas[i]=0;

         criterio=equipos-1;
         while (criterio > 1) /* mientras quede más de una fila para intercambiar*/
         {  asignar_mejor_fixture(fixture_aux,m,equipos);
            fila1= rand() % (equipos-1); /* se obtiene la primera fila a intercambiar*/
                do{
                    fila2= rand() % (equipos-1); /* se obtiene la segunda fila distinta a la primera*/
                }while(fila2 == fila1);

                if ( filas[fila1] ==0 && filas[fila2]==0) /* si ninguna se ha intercambiado antes*/
                {
                        for (j=0;j<equipos;j++)
                        {   fixture_aux[fila1][j]=m[fila2][j];
                            fixture_aux[fila1+equipos-1][j]=m[fila2+equipos-1][j];
                            fixture_aux[fila2][j]=m[fila1][j];
                            fixture_aux[fila2+equipos-1][j]=m[fila1+equipos-1][j];
                         }

                        if (evaluar_fixture(equipos,distancia,m) >
                        evaluar_fixture(equipos,distancia,fixture_aux)  &&
                        evaluar_factibilidad(equipos,fixture_aux)!=0) /* si el cambo mejora */
                        {
                          asignar_mejor_fixture(m,fixture_aux,equipos);
                          il(equipos,m,distancia);
                         /* se bloquean las filas cambiadas */
                          filas[fila1]=1;
                          filas[fila2]=1;
```

```
                        criterio=criterio-2; /* hay dos filas menos que intercambiar */
                        }

                }
 }
                        liberar_fixture(equipos,fixture_aux);
                        liberar_poligono(filas);
}


/* INTERCAMBIO DE LOCALIAS*/

void il_hill(int equipos,int **m, int **distancia)
{  int **fixture_aux;
   int fila,columna,anterior,i,j;
   fixture_aux=crear_fixture(equipos); /*se crea un fixture auxiliar*/

   for(fila=0;fila<equipos-1;fila++)
         for(columna=0;columna<equipos;columna++)
         {  if (fixture_aux[fila][columna] == 0) /*si no se han cambiado localía a esa celda*/
           {   anterior=evaluar_fixture(equipos,distancia,m);
              /* se cambia localía */
               m[fila][columna]*=-1;
               m[fila][abs(m[fila][columna])-1]*=-1;
               m[fila+equipos-1][columna]*=-1;
               m[fila+equipos-1][abs(m[fila][columna])-1]*=-1;

               if (evaluar_fixture(equipos,distancia,m)< anterior  &&
               evaluar_factibilidad(equipos,m)!=0) /* se evalúan los costos actuales y anteriores*/
               {    fixture_aux[fila][columna]=1;
                    fixture_aux[fila][abs(m[fila][columna])-1]=1;
                    fila=0; /* se comienza a cambiar desde el principio*/
                    columna=0;
                }
                Else /* se devuelve el cambio */
               {   m[fila][columna]*=-1;
                   m[fila][abs(m[fila][columna])-1]*=-1;
                   m[fila+equipos-1][columna]*=-1;
                   m[fila+equipos-1][abs(m[fila][columna])-1]*=-1;
                }
           }
         }
    liberar_fixture(equipos,fixture_aux);
}
```

```
/* VARIACION AL CAMBIO DE RONDA */

void ir_2(int equipos,int **m, int **distancia)
{  int **fixture_aux;
   int *orden_filas;
   int i,j,criterio=0;
   orden_filas=crear_poligono(equipos-1); /* se crea un vector auxiliar */
   fixture_aux=crear_fixture(equipos); /* se crea un fixture auxiliar */

    while (criterio < 120) /* numero medio entre las instancias */
          {  iniciar_poligono(equipos-1,orden_filas); /*se inicia el vector con un orden de filas*/
                for (i=0;i<equipos-1;i++)
                {   for (j=0;j<equipos;j++)
                    {      /* se aplica el orden de filas al fixture */
                           fixture_aux[i][j]=m[orden_filas[i]-1][j];
                           fixture_aux[i+equipos-1][j]=m[(orden_filas[i]-1)+equipos-1][j];
                     }
                 }
                if (evaluar_fixture(equipos,distancia,m) >
                evaluar_fixture(equipos,distancia,fixture_aux)
                && evaluar_factibilidad(equipos,fixture_aux)!=0)
                {   asignar_mejor_fixture(m,fixture_aux,equipos);

                 }
                Else  criterio++;
           }
                liberar_fixture(equipos,fixture_aux);
                liberar_poligono(orden_filas);
}
```

```
/* INTERCAMBIO DE LOCALIA CON SIMULATED ANNEALING*/

void il(int equipos,int **m, int **distancia)
{   int fila,columna,anterior,termino=0;
    double probabilidad;
    double t_inicial=800.0; /* Se define la temperatura inicial */

        do
        {  fila=rand()% (equipos-1);  se elije aleatóriamente la celda a cambiar localía*/
           columna= rand()% equipos;
                anterior=evaluar_fixture(equipos,distancia,m); /* se obtiene el costo anterior*/
               /* Se produce el cambio */
                m[fila][columna]*=-1;
                m[fila][abs(m[fila][columna])-1]*=-1;
                m[fila+equipos-1][columna]*=-1;
                m[fila+equipos-1][abs(m[fila][columna])-1]*=-1;

                if (evaluar_factibilidad(equipos,m)!= 0 )
                {  if (evaluar_fixture(equipos,distancia,m)> anterior )
                   {    /* Si el costo es mayor que el anterior, se calcula la probabilidad */
                        probabilidad= (double)rand()/ (double)RAND_MAX;
                        if ( probabilidad >
                        sim_ann(anterior,evaluar_fixture(equipos,distancia,m),t_inicial))
                        {       /*Si el numero aleatorio esta fuera de la probabilidad*/
                                /* se devuelve el movimiento*/
                                m[fila][columna]*=-1;
                                m[fila][abs(m[fila][columna])-1]*=-1;
                                m[fila+equipos-1][columna]*=-1;
                                m[fila+equipos-1][abs(m[fila][columna])-1]*=-1;

                        }else ir_2(equipos,m,distancia);
                      }else ir_2(equipos,m,distancia);

                    }else{
                         m[fila][columna]*=-1;
                         m[fila][abs(m[fila][columna])-1]*=-1;
                         m[fila+equipos-1][columna]*=-1;
                         m[fila+equipos-1][abs(m[fila][columna])-1]*=-1;

                    }
                        t_inicial= 0.99*t_inicial; /* se decrementa la temperatura */
                        termino++;
                }while (termino<700);

  }
```

A.8.6.- Proceso de Selección de Padres (seleccion.c)

```
#include "cabecera.h"

struct padres * torneo(struct agente *pprimero,int cantidad, int cardinalidad)
{int i,aleatorio,contador=1;
 struct padres *selec_padres;
 struct agente *busca;
 int *grupo;
 selec_padres=malloc(sizeof(struct padres*));

 (*selec_padres).padre1=NULL;
 (*selec_padres).padre2=NULL;


 grupo=crear_poligono(cantidad);

 for (i=0;i<cantidad;i++) grupo[i]=0;
   for (i=0;i<cardinalidad;i++)
   {
          do{
                aleatorio=(rand()%cantidad)+1;
          }while(grupo[aleatorio-1] !=0);

          grupo[aleatorio-1]=1;

           busca=pprimero;
           contador=1;

           while(contador!=aleatorio)
                {
                  busca=(*busca).siguiente;
                  contador++;

                }


                        if ((*selec_padres).padre1 == NULL)    (*selec_padres).padre1=busca;
                        else
                                if ((*selec_padres).padre2 == NULL)
                                        if ((*(*selec_padres).padre1).fitness > (*busca).fitness)
                                        {
                                                (*selec_padres).padre2=(*selec_padres).padre1;
                                          (*selec_padres).padre1=busca;

                                        }
                                        else (*selec_padres).padre2=busca;
                                else
```

```
                                        if ( (*busca).fitness < (*(*selec_padres).padre1).fitness )
                                        {       (*selec_padres).padre2=(*selec_padres).padre1;
                                                (*selec_padres).padre1=busca;
                                        }
                                        else
                                        if ( (*busca).fitness < (*(*selec_padres).padre2).fitness
)(*selec_padres).padre2=busca;


 }
 liberar_poligono(grupo);
 return(selec_padres);
}
```

A.8.7.- Recombinación (recombinacion.c)

```
#include "cabecera.h"

struct agente * recombinacion(struct agente *padre1,struct agente *padre2, int equipos,int
**distancia)
{int **hijo1;
 int **hijo2;
 int i,j,res1,res2,factible1,factible2;
 struct agente *hijo;
 int * poligono;


 /** Se crea en memoria la estructura hijo **/
 hijo=malloc(sizeof(struct agente*));
 /** Se crea las matrices para dos hijos **/
 hijo1=crear_fixture(equipos);
 hijo2=crear_fixture(equipos);

/** Se traspasan a los dos hijos lo genes repetidos en los padres **/
 for (i=0;i<equipos-1;i++)
           for (j=0;j<equipos;j++)
           { if (abs((*padre1).matriz[i][j])== abs((*padre2).matriz[i][j]))
                    { hijo1[i][j]=abs((*padre1).matriz[i][j]);
                      hijo1[i+equipos-1][j]=abs((*padre1).matriz[i][j]);

                     hijo2[i][j]=abs((*padre1).matriz[i][j]);
                     hijo2[i+equipos-1][j]=abs((*padre1).matriz[i][j]);
                    }
             }
```

```
                        /** Se manda a completar los dos hijos, y se recibe su factibilidad**/
                        factible1=completa_hijo(hijo1,equipos);
                        factible2=completa_hijo(hijo2,equipos);


                        /** Si no son factibles, se le asigna un fitness muy alto**/
                        /** Si son factibles, se les asigna localías **/
                        if (factible1==0)
                        {       asignar_localias(equipos,hijo1);
                                res1=evaluar_fixture(equipos,distancia,hijo1);
                        }else res1=1000000;
                        if (factible2==0)
                        {       asignar_localias(equipos,hijo2);
                                res2=evaluar_fixture(equipos,distancia,hijo2);
                        }else res2=1000000;


        /** Se comparan los dos hijos y el major es devuelto **/
        if (res1 <= res2 )
        {       (*hijo).matriz=hijo1;
                (*hijo).fitness=res1;
                (*hijo).siguiente=NULL;
                 liberar_fixture(equipos,hijo2);
        }
        else
        {       (*hijo).matriz=hijo2;
                (*hijo).fitness=res2;
                (*hijo).siguiente=NULL;
                 liberar_fixture(equipos,hijo1);
        }

return(hijo);

}

/**FUNCION COMPLETAR HIJO **/

int completa_hijo(int **hijo,int equipos)
{  int i,j,rival,espera,repite=0,desechar=0;

  do{
  repite=0;

  for (i=0;i<equipos-1;i++)
 {
          for (j=0;j<equipos;j++)
        {
                  if ( hijo[i][j] == 0) /** verifica si la casilla está vacía**/
                { espera=0;
```

```
            /** Asigna valores de equipos a la celda hasta que uno sea factible, o deja un cero**/
                do{
                        espera++;
                        rival= (rand() % equipos) + 1;
                        if (revisa_fila(hijo,rival,i,equipos)==0 &&
                        revisa_columna(hijo,rival,j,equipos)==0 && rival != (j+1))
                        {       hijo[i][j]=rival;
                                hijo[i][rival-1]=j+1;
                                hijo[i+equipos-1][j]=rival;
                                hijo[i+equipos-1][rival-1]=j+1;
                        }
                        }while (hijo[i][j]==0 && espera<10); /** 10 oportunidades de llenar **/
                 }
          }
             if (revisa_fila(hijo,0,i,equipos)==1)
                  {  repite=1;
                    desechar++;
                    borrar_fila(hijo,i,equipos);
                   }
    }
}while (repite==1 && desechar<20); /** 20 oportunidades de llenar fila **/
return(repite);
}

/** FUNCION QUE REVISA CELDAS VACIAS EN LAS FILAS **/
int revisa_fila(int **hijo,int rival,int fila, int equipos)
{int j,esta=0;
        for (j=0;j<equipos;j++)
                if ( hijo[fila][j]== rival)
                { esta=1;

                }

 return(esta);
}

/** FUNCION QUE REVISA CELDAS VACIAS EN COLUMNAS **/

int revisa_columna(int **hijo,int rival,int columna, int equipos)
{int i,esta=0;
        for (i=0;i<equipos-1;i++)
                if ( hijo[i][columna]== rival)
                { esta=1;
                }


 return(esta);
}
```

A.8.8.- Función Objetivo (funcion_objetivo.c)

```
#include "cabecera.h"

int evaluar_fixture(int equipos,int** distancia, int**m)
{int i,j,recorrido=0;

  for (i=0;i<equipos;i++) /** se contabiliza la primera fila de la matriz **/
          if (m[0][i]<0)  recorrido += distancia[i][abs(m[0][i])-1];

  for (j=0;j<equipos;j++) /** se contabilizan desde la segunda a la penúltima fila **/
          for (i=1; i<2*(equipos-1);i++)
          { /** primer caso, ronda r visita, r+1 local **/
            if (m[i-1][j]<0  && m[i][j]>0) recorrido += distancia[j][abs(m[i-1][j])-1];
           /** segundo caso, ronda r local, r+1 visita **/
            if (m[i-1][j]>0  && m[i][j]<0) recorrido += distancia[j][abs(m[i][j])-1];
           /** tercer caso, ronda r visita, r+1 visita **/
            if (m[i-1][j]<0  && m[i][j]<0) recorrido += distancia[abs(m[i-1][j])-1][abs(m[i][j])-1];
          }

   for (i=0;i<equipos;i++) /** se contabiliza ultima fila **/
            if (m[(2*(equipos-1))-1][i]<0)  recorrido += distancia[i][abs(m[(2*(equipos-1))-1][i])-1];

    if (evaluar_factibilidad(equipos,m) == 0) recorrido=1000000;

    return(recorrido);
}
/** FUNCION PARA EL CONTROL DE RESTRICCION U=3 **/
int evaluar_factibilidad(int equipos,int **m)
{int i,j,acumulador,lov, error=0;

   for (j=0;j<equipos;j++)
   {       acumulador = 1;
           lov = m[0][j];
             for (i=1;i< (2*(equipos-1));i++)
             {       if ((m[i][j]>0 && lov>0) || (m[i][j]<0 && lov<0)) acumulador++;
                     if ((m[i][j]<0 && lov>0) || (m[i][j]>0 && lov<0)) acumulador=1;
                          if (acumulador>3) /* CONTROL DE RESTRICCION U=3 */
                          {    error++;
                               acumulador=1;
                          }
                      lov=m[i][j];
             }
   }
   if (error==0) return(1);
   else return(0);
}
```