CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA, BAJA CALIFORNIA

PROGRAMA DE POSGRADO EN CIENCIAS EN CIENCIAS DE LA COMPUTACIÓN

Aplicación de algoritmos genéticos al problema PATS de auto-ensamblado de ADN

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Ema Lucía Lozano Guzmán

Ensenada, Baja California, México

2014

Tesis defendida por

Ema Lucía Lozano Guzmán

y aprobada por el siguiente comité

Dr. Israel Marck Martínez Pérez
Director del Comité

Dr. Carlos Alberto Brizuela Rodríguez
Miembro del Comité

Dr. Hugo Homero Hidalgo Silva
Miembro del Comité

Dr. David Hilario Covarrubias Rosales
Miembro del Comité

Dra. Ana Isabel Martínez García
Coordinador del Programa de
Posgrado en Ciencias de la Computación

Dr. Jesús Favela Vara
Director de Estudios de Posgrado

Septiembre, 2014

Resumen de la tesis que presenta Ema Lucía Lozano Guzmán como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Aplicación de algoritmos genéticos al problema PATS de auto-ensamblado de ADN

Resumen elaborado por:

Ema Lucía Lozano Guzmán

El problema "*Patterned self-Assembly Tile set Synthesis*" (PATS) del modelo de auto-ensamblado de moléculas de ADN, se ocupa de encontrar un conjunto mínimo de tipos de moléculas que sean capaces de auto-ensamblar un patrón P de dos dimensiones previamente definido. Para realizar el auto-ensamblado, se utiliza el modelo abstracto de ensamble de mosaicos (aTAM por sus siglas en inglés), donde cada mosaico representa una molécula de ADN con doble cruzamiento, cuyos extremos pegajosos se especifican como fuerza de pegado en cada uno de los lados del mosaico. El problema PATS pertenece al conjunto de problemas NP-difícil. Hasta ahora la literatura le relaciona tres trabajos que lo abordan aplicando diferentes heurísticas. Aunque estos enfoques obtienen reducciones significativas del conjunto de tipos de mosaicos, el problema PATS queda abierto para la aplicación de otros algoritmos. En esta investigación se presenta un algoritmo genético para solucionar este problema. La representación del individuo y operadores genéticos adoptan una estructura matricial. Experimentos computacionales consideran que esta propuesta encuentra una solución que reduce el conjunto de tipos de mosaicos, superando los resultados publicados en trabajos del estado del arte.

Palabras Clave: **Algoritmos genéticos, Auto-ensamblado de ADN, PATS.**

Abstract of the thesis presented by Ema Lucía Lozano Guzmán as a partial requirement to obtain the Master of Science degree in Master in Sciences in Computer Science.

Application of genetic algorithms to the PATS problem of DNA self-assembly

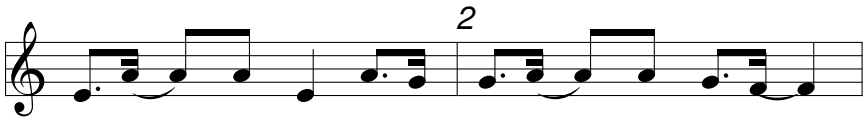
Abstract by:

Ema Lucía Lozano Guzmán

The Patterned self-Assembly Tile set Synthesis (PATS) problem determines a minimal set of tile types that self-assembles a given two-dimensional pattern. The self-assembly is performed by using the abstract Tile Assembly Model (aTAM), where each tile represents a double crossover DNA molecule whose sticky ends are specified as a bonding force in each side of the tile. The PATS problem belongs to the set of NP-hard problems. So far this problem has been addressed in three studies where different algorithm were applied. While these approaches achieve significant reductions of the set of tile types needed to form a pattern, the PATS problem remains open for the application of other strategies. In this work, a genetic algorithm to solve the PATS problem is presented. A key feature of this approach, the chromosome and genetic operators adopt a matrix structure. Computational experiments were developed to find a solution that reduces the set of tile types, surpassing the results reported in the literature.

Keywords: **Genetic algorithmic, DNA Self-assebly, PATS.**

Dedicatoria



Agradecimientos

A mi familia.

A Trisha y Luz por mantenerme cuerda.

A mis personas especiales por estar conmigo.

A mis amigos por frecuentarme y estar ahí para mí.

A mi comité de tesis por guiarme en esta etapa.

Al Centro de Investigación Científica y de Educación Superior de Ensenada.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por brindarme el apoyo económico para realizar mis estudios de maestría.

Tabla de contenido

Página

Tabla de contenido (continuación)

Tabla de contenido (continuación)

Lista de figuras

Figura Página

Lista de figuras (continuación)

Lista de figuras (continuación)

Figura Página

Lista de tablas

Capítulo 1. Introdución

El auto-ensamblado es el proceso por el cual una colección de componentes relativamente simples, a partir de un estado desorganizado, de forma espontánea y sin guía externa se une para formar estructuras más complejas (Adleman, 1994). El proceso es guiado únicamente por las interacciones locales entre los componentes, que normalmente siguen un conjunto básico de reglas. A pesar de la naturaleza aparentemente simplificada del auto-ensamblaje, su poder puede ser aprovechado para formar estructuras de increíble complejidad. De hecho, los sistemas de auto-ensamblado abundan en la naturaleza, lo que resulta en todo, desde la estructura cristalina delicada de los copos de nieve a muchos de los componentes estructurales y funcionales de los sistemas biológicos. Más allá de las propiedades matemáticamente interesantes de los sistemas de auto-ensamblado, estos sistemas han sido reconocidos como un modelo excelente para la fabricación de estructuras artificiales a escala nanométrica (Doty *et al.*, 2012). Con el fin de manipular con precisión y organizar la materia a escala de átomos y moléculas individuales, varios sistemas artificiales de auto-ensamblaje han sido diseñados (Winfree *et al.*, 1998; Fujibayashi *et al.*, 2008; Winfree y Bekbolatov, 2003).

Uno de los primeros modelos diseñados para el auto-ensablado de ADN es el "*abstract Tile Assembly Model*" (aTAM), fruto del trabajo de Winfree en su tesis doctoral (Winfree *et al.*, 1998). Las unidades fundamentales del modelo aTAM son mosaicos de ADN (*tiles*) donde en cada uno de sus lados tiene una fuerza de pegado, con diferentes pesos. Los mosaicos de ADN se pegan entre sí sólo si los lados adyacentes son complementarios. Winfree *et al.* (1998) demostró experimentalmente que este modelo se puede implementar en un sistema molecular, donde cada mosaico es una molécula de ADN con doble cruzamiento (DX), con cuatro extremos pegajosos (*sticky ends*) que representan la fuerza de pegado de cada lado del mosaico de ADN. La estabilidad de los enlaces entre moléculas de ADN depende fuertemente de un parámetro externo: la temperatura presente en el medio.

En el modelo aTAM cada extremo pegajoso tiene diferente fuerza de pegado, que depende de la cantidad de enlaces de hidrógeno con los que cuente. Cada mosaico de ADN

tiene orientación en cada uno de sus lados, por esto no es posible rotarlo ni reflejarlo. El diseño de un mosaico de ADN se basa en asignar la fuerza de pegado a cada uno de sus extremos pegajosos. Además, puede existir una infinidad de ejemplares en una superficie de dos dimensiones. Los mosaicos de ADN se van pegando uno a uno en una estructura que va creciendo conforme al ensamblado. Cuando los extremos pegajosos de un mosaico de ADN coinciden con los de la estructura, se ensambla. Se supone que los extremos pegajosos actúan de forma cooperativa. Es decir, si los extremos pegajosos son compatibles y son más de uno los que se pegan, entonces la suma de dichos extremos pegajosos es la cantidad que debe ser suficiente en relación a la temperatura para mantener la unión estable.

Las hipótesis biológicas para que dichas moléculas de ADN se comporten de acuerdo al modelo aTAM son las siguientes:

- Las moléculas de doble cruzamiento pueden ser diseñadas en el laboratorio para auto-ensamblarse en cristales bidimensionales. Esto ha sido demostrado en un sistema experimental (Winfree *et al.*, 1998).
- La fuerza de pegado puede ser implementada diseñando extremos pegajosos con energía específica de hibridación. El poder de hibridación del ADN depende principalmente del número de pares de bases. Así, por ejemplo, extremos pegajosos más largos pueden ser usados para representar pegamentos más poderosos.
- Cuando dos extremos pegajosos son complementarios, las moléculas de doble cruzamiento se hibridan. Evidencia empírica de esto se presenta en el trabajo de Winfree *et al.* (1998).

El modelo aTAM involucra problemas del tipo NP-Difícil, como el problema "*Patterned self-Assembly Tile set Synthesis*" (PATS), el cuál se ocupa de encontrar el conjunto mínimo de mosaicos de ADN que se auto-ensambla en un determinado patrón en una red rectangular en un plano de 2-D, cuyo patrón está dado por colores (blanco y negro). Este problema fue introducido en (Ma y Lombardi, 2008). El problema PATS pertenece

al conjunto de problemas NP-difícil (Czeizler y Popa, 2012), por lo que es viable aplicar heurísticas para intentar obtener una solución óptima. La optimalidad se define en encontrar la mínima cardinalidad del conjunto de tipos de mosaicos.

En este trabajo se propone un algoritmo genético para solucionar el problema PATS. Los algoritmos genéticos proveen un campo de búsqueda amplio dada su naturaleza. Están inspirados en la selección natural, donde existe una población de individuos en un entorno con recursos limitados. La competencia por los recursos hace que la selección de los individuos más aptos y mejor adaptados al medio ambiente sobrevivan. Estos individuos actúan como semillas para la generación de nuevos individuos a través de la recombinación y mutación. Los nuevos individuos tienen su aptitud evaluada por una función objetivo y compiten para la supervivencia. Con el tiempo, la selección natural provoca un aumento de la aptitud de la población y converge a un individuo más apto (Eiben y Smith, 2003). Hasta nuestro conocimiento, en la literatura aún no existe una aplicación de este tipo de algoritmos al problema PATS, por lo cual se propone abordar este problema con un enfoque evolutivo.

1.1. Antecedentes y motivación

El problema PATS es relativamente nuevo, por ello existen pocos trabajos que lo abordan. Hasta ahora únicamente se le relacionan tres, los cuales tratan de encontrar mejores soluciones aplicando diferentes estrategias. Según la literatura, el primer algoritmo que se le aplicó al problema PATS es el voraz (Ma y Lombardi, 2008) . Este se caracteriza por escoger la mejor solución posible en cada paso local con la esperanza de llegar a una solución óptima. El segundo es conocido como *Branch-and-Bound* (Göös y Orponen, 2010), el cual utiliza una estructura de árbol, la recorre de forma sesgada en busca de la mejor solución. El último algoritmo aplicado al problema PATS, según el estado del arte, es una combinación de dos estrategias, *Partition-Search* con *Branch-and-Bound* (Lempiäinen *et al.*, 2011). En el capítulo 2 se describe a detalle cada uno de ellos.

El motivo inicial de este trabajo es generar un algoritmo genético que sea capaz de encontrar un conjunto reducido de tipos de mosaico, ya que al hacerlo se reduce consi-

derablemente el porcentaje de error de auto-ensamble del patrón obtenido (Chen *et al.*, 2004), por lo cual es de suma importancia acercarse a la solución óptima del problema PATS. Anteriormente se han presentado tres algoritmos diferentes en los que se obtiene reducciones significativas del conjunto de tipos de mosaicos, sin embargo, el problema queda abierto para la aplicación de otras estrategias, como algoritmos genéticos, para intentar obtener mejores resultados.

1.2. Planteamiento del problema

En este trabajo de investigación se pretende abordar el problema PATS de auto-ensamblado de ADN, aplicando un algoritmo genético. Para ello es necesario desarrollar un algoritmo genético que permita reducir el conjunto de tipos de mosaicos, por lo que se requiere diseñar la representación del individuo, generar la inicialización de la población, diseñar e implementar operadores genéticos y el mecanismo de evaluación de la función objetivo. A la fecha se han propuesto únicamente tres algoritmos que obtienen un conjunto reducido de tipos de mosaicos, no obstante, el problema queda abierto para ser explorado por otras heurísticas. Dada la naturaleza de búsqueda que usan los algoritmos genéticos, es viable aplicarlo al problema PATS para intentar obtener resultados comparables a los logrados por métodos del estado del arte.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar e implementar un algoritmo genético que resuelva el problema PATS de auto-ensamblado de ADN, para disminuir el error de pegado entre mosaicos durante el proceso de auto-ensamblado de un patrón dado.

1.3.2. Objetivos específicos

1. Diseñar la representación del individuo.
2. Diseñar los operadores de cruce y mutación.

3. Diseñar la función objetivo, que minimice el conjunto de tipos de mosaicos.
4. Codificar el algoritmo genético, la inicialización de la población, la representación del individuo y los operadores genéticos.
5. Combinar los distintos operadores genéticos para encontrar la mejor configuración.
6. Realizar simulaciones con los diferentes patrones (casos de prueba).
7. Comparar los resultados obtenidos del algoritmo genético con los resultados existentes de los trabajos previos.

1.4. Metodología

Para cumplir con los objetivos propuestos, se siguió la metodología que se describe a continuación. Para la inmersión en el problema PATS se revisó el trabajo de Ma y Lombardi (2008), en el cual se introduce el problema. Se realizó una revisión de la literatura, encontrando trabajos que lo abordan aplicando distintos algoritmos (Ma y Lombardi, 2008; Göös y Orponen, 2010; Lempiäinen *et al.*, 2011). Como solución alternativa, se propuso la aplicación de un algoritmo genético para el problema PATS. Se inició con el diseño de la representación del individuo, parte que requirió de creatividad para representar las características que se desean evolucionar. El enfoque que se utilizó fue representar al individuo como una imagen de pixeles, donde cada pixel es un mosaico y la imagen es el patrón dado. El siguiente paso para el diseño del algoritmo fue la generación de la población inicial, donde se determinaron las restricciones de las cuales el individuo estuvo sujeto para formar parte de la población. Estas restricciones fueron básicamente dos: que el individuo no tuviera errores de pegado entre los mosaicos y que el ensamblaje formará el patrón dado. Después se prosiguió con el diseño de los operadores genéticos. Se utilizaron tres operadores de cruce y el diseño se tomó de la literatura. Los tres operadores de mutación fueron diseñados con base en una función que genera mosaicos nuevos partiendo de otro. Posteriormente, se realizó el mecanismo de evaluación de la función objetivo, donde se determinó que la aptitud estaría dada en base a los tipos de mosaicos en el individuo evaluado. Una vez diseñado el algoritmo genético, se prosiguió con la implementación

del mismo. La fase de experimentación se efectuó mediante simulaciones computacionales del algoritmo genético. Aquí se combinaron los diferentes operadores de cruce y mutación en busca de una configuración de parámetros que obtuviera buenos resultados para cada caso de prueba de la literatura. Por último, estos resultados fueron analizados y comparados con aquellos reportados en los métodos del estado del arte.

1.5. Organización de la tesis

A continuación se explica la organización del presente trabajo: iniciamos en el Capítulo 2 donde se introduce el marco teórico, que incluye los fundamentos de biología molecular, que es el pilar fundamental del cómputo biomolecular. También incluye una descripción de los fundamentos de computación y de cómputo biomolecular, introduciendo antecedentes y los distintos modelos, haciendo énfasis en el modelo de auto-ensamblado de ADN. En el Capítulo 3 se presenta el diseño del algoritmo genético para el problema PATS. Se describe la representación del individuo, el diseño de operadores genéticos y la inicialización de la población. En el Capítulo 4 se presentan los detalles de la simulación del algoritmo genético y los resultados obtenidos. Finalmente en el Capítulo 5 se presentan las conclusiones y algunas ideas de trabajo futuro.

Capítulo 2. Marco teórico

En este capítulo se abordan conceptos y definiciones de los fundamentos de la biología molecular. También se incluyen los fundamentos de computación y cómputo biomolecular. Estas tres secciones constituyen el marco teórico de la presente investigación y son descritas a continuación.

2.1. Fundamentos de biología molecular

2.1.1. La célula

En biología la unidad mínima funcional de los seres vivos es la célula. Los organismos de dichos seres pueden ser unicelulares o pluricelulares; tal y como lo describe su nombre, los organismos unicelulares estan formados por una sola célula y los pluricelulares por múltiples de ellas. Se dice que la estructura de la célula está compuesta por citoplasma en su interior y un recubrimiento de membrana que a su vez está compuesta principalmente de lípidos. En el citoplasma se almacenan organelos que son los encargados de la supervivencia de la celula.

Existen dos tipos de células (Lewin, 2008): procariotas y eucariotas (Figura 1). Las células eucariotas presentan un citoplasma organizado en compartimentos, con orgánulos que se encuentran separados o interconectados, limitados por membranas biológicas que tienen la misma naturaleza que la membrana plasmática. El núcleo es el más notable y característico de los compartimentos en que se divide el protoplasma, es decir, la parte activa de la célula. En el núcleo se encuentra el material genético en forma de cromosomas, este es el encargado de dar toda la información necesaria para que se lleve a cabo todos los procesos tanto intracelulares como fuera de la célula. En el protoplasma se distinguen tres componentes principales, la membrana plasmática, el núcleo y el citoplasma. Las células eucariotas están dotadas en su citoplasma de un citoesqueleto complejo, muy estructurado y dinámico, formado por microtúbulos y diversos filamentos proteicos.

A diferencia de la célula eucariota, la procariota es pequeña y menos compleja, contiene ribosomas pero carece de sistemas de endomembranas y de núcleo celular. Por

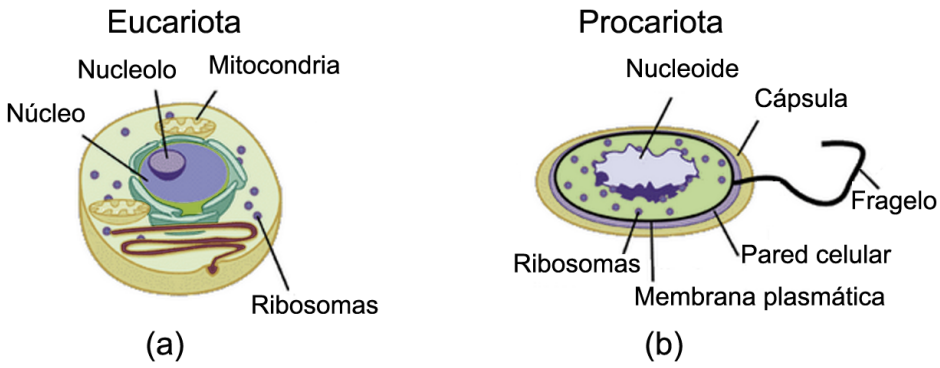


Figura 1: Tipos de células (basado en Navarta, A. (2012)): (a) eucariota, (b) procariota.

lo general podría decirse que las células procariotas carecen de citoesqueleto y por ello poseen el material genético en el citosol. Ambos tipos de células contienen ácido desoxiribonucleico (ADN), un polinucleótido que reside en los cromosomas y que su función principal es la replicación. Para que el ADN pueda expresarse como proteínas, antes es necesario convertirlo en moléculas de ácido ribonucleico (ARN), que se usan para la síntesis de proteínas.

2.1.2. Ácido desoxirribonucleico (ADN)

Los ácidos nucleicos son polímeros que se forman por la repetición de nucleótidos, unidos por enlaces fosfodiéster. Un nucleótido es una molécula que esta formada por una base nitrogenada, que puede ser purina o pirimidina; la única diferencia entre ellas es la presencia de un azúcar (ya sea ribosa o desoxirribosa) y un grupo fosfato. Los ácidos nucleicos almacenan la información genética de los organismos vivos y son los responsables de la transmisión hereditaria. Existen dos tipos básicos, el ADN y el ARN.

El ácido desoxirribonucleico, mejor conocido como ADN (Figura 2) es una molécula larga, en escalera, que forma una doble hélice. Cada cadena de la hélice es una molécula lineal formada por subunidades llamados nucleótidos. En el ADN hay cuatro nucleótidos diferentes, cada nucleótido tiene una de las cuatro bases nitrogenadas: las bases son adenina (A), guanina (G), citosina (C) y timina (T). Estas cuatro bases forman el alfabeto genético, o código genético, que en distintas combinaciones especifica finalmente la secuencia de los aminoácidos de las proteínas.

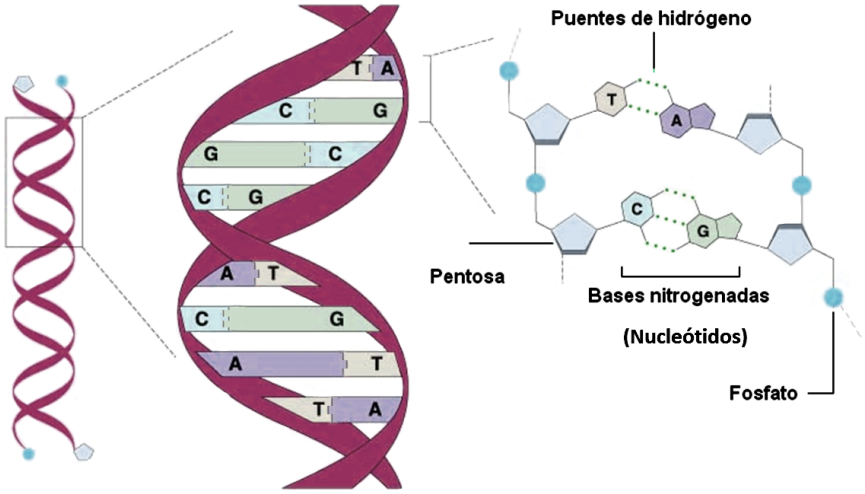


Figura 2: Estructura del ADN, modelo de Watson-Crick (basado en Becerril, J. (2013)).

Uno de los grandes descubrimientos en biología del siglo XX fue llevado a cabo por Watson y Crick (1953), quienes establecieron que las dos cadenas de ADN son complementarias entre si, esto es, que los escalones de la escalera de doble hélice constan de los pares de bases A con T y G con C por medio de enlaces no covalentes generados por la interacción entre una base nitrogenada purina con una pirimidina. Los enlaces no covalentes son puentes de hidrógeno, los cuales varian dependiendo del par de bases nitrogenadas; las bases G y C generan tres puentes de hidrógeno, mientras que A y T generan únicamente dos (Lewin, 2008). Esta relación sirve de base para la replicación del ADN y para la expresión genética. Durante ambos procesos el ADN sirve de molde para la síntesis de moléculas complementarias.

2.1.3. Hibridación de ADN

La hibridación de ADN es un proceso de unión entre dos cadenas complementarias de ADN (Lewin, 2008). El método usado para realizar este proceso inicia separando las hebras de ADN, elevando la temperatura para romper los puentes de hidrógeno de las bases (desnaturalización). Esto ocurre cuando la temperatura alcanza alrededor de 90° (dependiendo de la longitud de las secuencias de ADN), y se conoce como temperatura de fusión. Cuando una solución de ADN ha sido calentada y se enfría lentamente, se produce la rehibridación dando la estructura inicial. Este fenómeno ocurre sólo si las se-

cuencias de bases son complementarias. Este es un método muy versátil que permite estudiar el grado de relación genética entre dos ácidos nucleicos.

2.1.4. Replicación de ADN

La operación de replicación tiene lugar cuando copias idénticas de ADN son creadas en un proceso iterativo llamado PCR (*Polymerase Chain Reaction*) que se basa en el uso de la enzima ADN Polimerasa. Dada una molécula dsADN que sirve como plantilla, oligos cortos iniciadores (*primers*) que contienen las primeras bases de los extremos 5' de cada ssADN que compone la plantilla, y la presencia de suficientes nucleótidos, la ADN polimerasa extiende los *primers* cuando están hibridados en la plantilla (McPherson y Møller, 2000). Un ciclo de PCR consiste de los siguientes pasos:

1. Calentamiento para desnaturalizar las plantillas.
2. Enfriamiento para la hibridación de los *primers* en los extremos 5' de cada ssADN de la plantilla.
3. La ADN polimerasa extiende los *primers* añadiendo sucesivamente nucleótidos en sus extremos 3' (siempre en la dirección 5' $\rightarrow$ 3').

Al final de n iteraciones se producen 2^n copias de una plantilla (Figura 3).

2.1.5. Moléculas de ADN con doble cruzamiento

Las moléculas de ADN con doble cruzamiento contienen dos sitios de cruce entre dominios helicoidales, la distancia entre estos puntos deben especificarse como múltiplos de 2 (modulo 2), a fin de evitar el estrés torsional en ellas. Dada la estructura de estas moléculas se dice que son altamente estables (Fu y Seeman, 1993). Como se conocen en la literatura, existen dos grandes clases: las paralelas y las antiparalelas. En cómputo biomolecular en el modelo de auto-ensamblado de ADN únicamente se utilizan las antiparalelas. Esta clase está constituida por dos tipos de moléculas: par e impar.

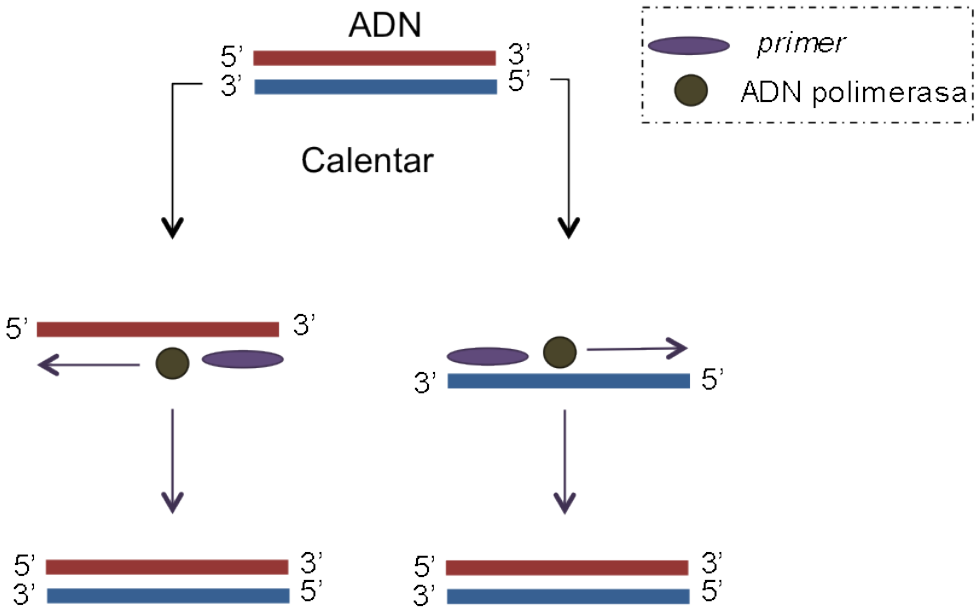


Figura 3: Replicación de ADN con PCR.

La molécula par "*Double-crossover Antiparallel Odd*" (DAO) está compuesta por cuatro cadenas sencillas de ADN que se complementan entre ellas con una longitud de 12.6 nm y una secuencia de 37 nucleótidos (Figura 4(a)). La molécula impar denominada "*Double-crossover Antiparallel Even*" (DAE) usa cinco fragmentos de cadenas sencillas de ADN que se complementan entre sí, con una longitud de 14.3 nm y una secuencia de 42 nucleótidos (Figura 4(b)). Ambas moléculas tienen cuatro extremos salientes, con una secuencia de cinco nucleótidos que pueden variar de longitud dependiendo del diseño de los mismos. El diseño se realiza con la finalidad de formar una macro-estructura mediante el proceso de auto-ensamblado de las mismas.

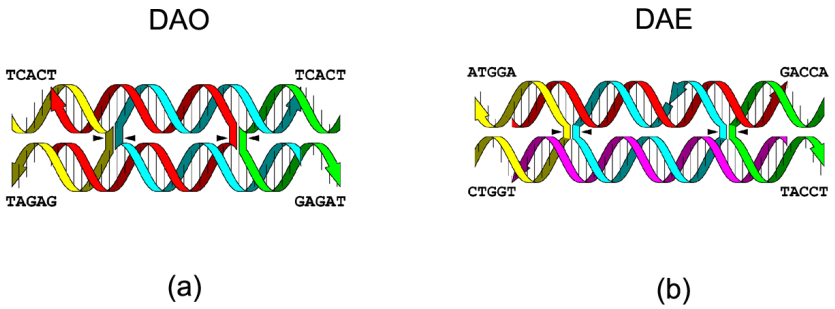


Figura 4: Estructura de moléculas de ADN con doble cruzamiento (basado en Rothemund *et al.* (2004)): (a) molécula DAO, (b) molécula DAE.

En el trabajo de Rothemund *et al.* (2004), se utilizaron las moléculas DAE y DAO para fabricar el patrón fractal del triángulo de Sierpinski mediante auto-ensamblado. Los autores fabricaron un autómata celular cuya regla de actualización calcula la función XOR binaria. Para el diseño de los extremos salientes se elaboraron 8 moléculas diferentes, 4 de ellas fueron DAO y 4 DAE. Se sometieron al proceso de auto-ensamblado en laboratorio experimental. Observaron que las tasas de error durante el proceso de auto-ensamblado parecen variar de 1 % a 10 %. Aunque, el crecimiento de los triángulos de Sierpinski contiene errores de pegado, la aportación más sobresaliente es el procedimiento donde muestran todos los mecanismos necesarios para la aplicación molecular de autómatas celulares en auto-ensamblado con moléculas de doble cruzamiento. Y demuestran que la ingeniería de auto-ensamblaje de ADN puede ser tratado como un sistema biomolecular capaz de implementar cualquier algoritmo deseado para tareas de computación o de la construcción de patrones.

2.2. Fundamentos de computación

En esta sección se introducen conceptos computacionales básicos que se manejan a lo largo de la presente tesis. Inicialmente, se describe el concepto de autómata finito y autómata celular. Posteriormente, se presentan conceptos de optimización y clasificación de métodos. Finalmente, se presenta la computación evolutiva, algoritmos genéticos y su aplicación.

2.2.1. Autómata de estados finitos

En la decada de 1930, A.M. Turing estudió una máquina abstracta con poder de cómputo. El objetivo de Turing era determinar el límite entre lo que puede y no puede hacer una máquina computadora. Las conclusiones de Turing no solo son aplicables a las máquinas de su tiempo sino también a las actuales. En las décadas de 1940 y 1950, otros investigadores se interesaron por unos tipos de máquinas más sencillas, conocidas como “autómatas finitos”. Estos autómatas, propuestos en un principio para modelar la función cerebral, dieron como resultado ser extremadamente útiles para otros diversos propósitos. Un autómata es definido como una unidad de procesamiento, que lee una cadena de entrada y la acepta o la rechaza.

2.2.1.1. Autómata de estados finitos determinístico

Un autómata de estados finitos determinístico (DFA) se define formalmente como una quíntupla $M = (\Sigma, S, \delta, s_0, F)$, donde Σ es un alfabeto, S es un conjunto finito de estados tal que $S \cap \Sigma = \varnothing$, $s_0 \in S$ es el estado inicial, F es el conjunto de estados finales, donde $F \subseteq S$, y δ es la función de transición $\delta : S \times \Sigma \to S$, donde la transición $\delta(s, a) = s'$. El tamaño de un autómata de estados finitos M, se denota como $\mid M \mid$ y esta dado por el número $\mid S \mid + \mid \delta \mid$.

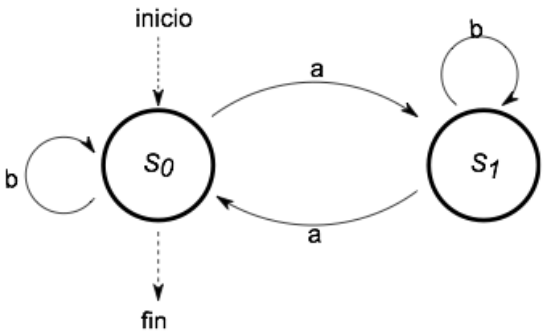


Figura 5: Autómata de estados finitos determinístico con dos estados (S_0, S_1) y dos símbolos (a,b).

Para ilustrar el funcionamiento de un DFA seguiremos un ejemplo sencillo. Considere un autómata de estados finitos M, un conjunto de estados $S = \{s_1, s_2\}$, un alfabeto $\Sigma = \{a, b\}$, un estado inicial s_0 y un conjunto de estados finales $F = \{s_0\}$, con reglas de transición δ, que son ilustradas en el grafo de la Figura 5.

Sea $x = aba$ la cadena de entrada. El cómputo inicia en el estado s_0; donde M lee el primer símbolo de la cadena de entrada, en este caso a, por lo que se transfiere al estado s_1, siguiendo la regla $\delta(s_0, a) = s_1$. Posteriormente, M lee el siguiente símbolo de la cadena de entrada, y se aplica la regla de transición $\delta(s_1, b) = s_1$; por lo que permanece en el mismo estado del autómata. Finalmente M lee el último símbolo, en este caso a, y se aplica la regla $\delta(s_1, a) = s_0$. Al no haber más símbolos en la cadena de entrada, M termina el procesamiento en el estado s_0, el cual acepta la cadena dado que $s_0 \in F$ (Hopcroft *et al.*, 2002).

2.2.2. Complejidad computacional

En complejidad computacional la reducción en tiempo polinomial proporciona un medio formal para demostrar que un problema es al menos tan difícil como otro. Por ejemplo, si $L_1 \leq_p L_2$, entonces L_1 no es más que un factor polinomial más difícil que L_2, donde la notación “menor o igual a” es un termino para indicar la reducción. Dentro de la clasificación de problemas se define NP como el conjunto de problemas que pueden ser resueltos en tiempo polinomial por una máquina de Turing no determinista. Sin embargo, NP contiene otros conjuntos tales como NP-completo y NP-difícil. Para saber si un problema pertenece a uno u otro conjunto, entonces suponemos un problema L, donde:

1. $L \in$ NP, y
2. $L' \leq_p L$ para cada $L' \in$ NP.

Si L satisface ambas propiedades, entonces pertenece a NP-completo. En cambio, si L satisface la propiedad 2, pero no necesariamente la propiedad 1, entonces L forma parte del conjunto NP-difícil. El conjunto NP-completo se considera como el conjunto de problemas más difíciles de NP.

2.2.3. Autómata celular

Un autómata celular se puede definir como la séxtupla (d, r, Q, N, V, f), donde d es la dimensión del autómata, la cual indica la organización espacial de la células. La posición de cada célula se expresa mediante un vector de Z_d. El parámetro r es el índice de localidad que marca el tamaño de la vencidad, es decir, indica el número de vecindad para cada célula. Q es el conjunto de estados; el estado en el que se encuentra cada célula. La variable N es un estado de Q, llamado estado quiestante, que indica la ausencia de actividad. V es un vector de vecindad que contiene r elementos distintos de Z_d, $V \subset (Z_d)^r$. Y por último, f es la función de transición o regla del autómata $f : Q_{r+1} \to Q$, donde $f(q_{i-r}(t-1), q_{i-r+1}(t-1), ..., q_{i+1}(t-1)) = q_i(t)$; siendo $q_i(t)$ el estado i en el tiempo t. Para cada célula f determina el estado que tendrá en la siguiente unidad de tiempo, en función de su estado actual y del estado de sus células vecinas.

2.2.3.1. Configuración de autómata celular

Se define la configuración de un autómata celular C (o palabra de C) a una función $\Sigma : Z_d \rightarrow Q$. La configuración de Σ es la palabra resultante de aplicar la regla del autómata a sus células. Como ejemplo, podemos tomar un autómata celular unidimensional circular con un radio de vecindad $r = 1$, dos estados Q dados por $\{0, 1\}$. Para este caso usaremos un autómata de diez células y dos funciones de transición basadas en lo siguiente:

1. Si ambos vecinos de una célula tienen el mismo estado, el estado de la célula a la que se aplica la función cambiará.
2. Si ambos vecinos de una célula tienen distinto estado, el estado de la célula a la que se aplica se mantendrá igual.

Suponga un valor inicial ($t = 0$) de 0001010011 (Figura 6). Las células sombreadas cambiaran su valor en la siguiente iteración. Para el tiempo $t = 1$, las células 2, 4, 5 y 6 cambian debido a la aplicación de la primera función de transición mientras que el resto de las células permanecen igual por la segunda función de transición. En $t = 2$, mediante la aplicación de las funciones de transición en el autómata, las células 1, 2, 5 y 7 son las que cambian. De esta forma el autómata celular va cambiando en el tiempo con base en las reglas definidas.

Uno de los autómatas célulares más conocidos es el del juego de la vida, el cual en realidad es un juego de cero jugadores, lo que quiere decir que su evolución está determinado por el estado inicial y no necesita ninguna entrada de datos posterior. El tablero de juego es una malla formada por cuadrados (células) que se extiende por el infinito en todas las direcciones. Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales. Las células tienen dos estados: están vivas o muertas (encendidas o apagadas). El estado de la malla evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente, de acuerdo a la función

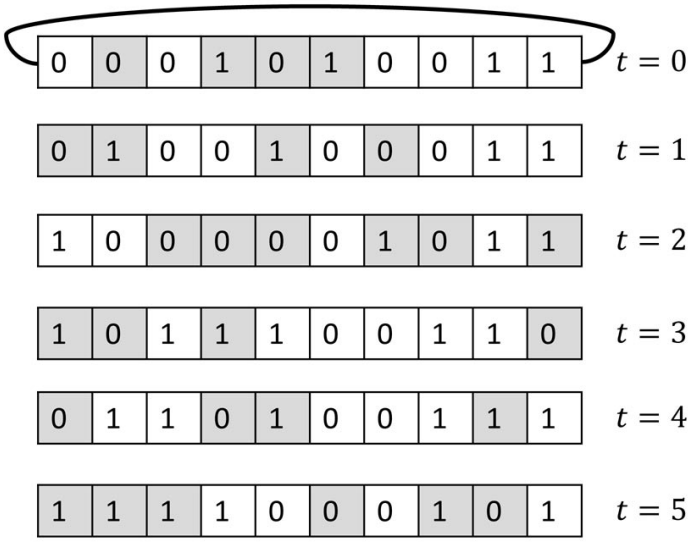


Figura 6: Autómata celular con estructura circular de 10 células con 6 iteraciones. Las células sombreadas cambian de valor en la siguiente iteración, de acuerdo a la función de transición 1.

$$f(q_0(t-1), q_1(t-1), ..., q_8(t-1)) = \begin{cases} \text{2 ó 3 si } \sum_{i=1}^{8} q_i(t-1) = 2, \\ \text{1 si } \sum_{i=1}^{8} q_i(t-1) = 3, \\ \text{0.} \end{cases} \quad (1)$$

Las transiciones dependen del número de células vecinas vivas:

- Una célula muerta con exactamente 3 células vecinas vivas nace (al turno siguiente estará viva).
- Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere o permanece muerta (por soledad o superpoblación).

Clasificación de autómatas celulares

De acuerdo a un estudio sobre el comportamiento dinámico de los autómatas celulares unidimensionales realizado por Wolfram (1983), estos pueden clasificarse en:

- **Tipo 1**: Todas las configuraciones evolucionan a un estado estable, independientemente de la configuración inicial. El sistema resulta predecible y los patrones desaparecen en el tiempo, por lo que no hay cambio en el estado final.

- **Tipo 2**: Las configuraciones tienden hacia estructuras periódicas. El cambio en un único valor en la situación inicial afecta únicamente a una región finita a su alrededor.
- **Tipo 3**: Presenta un comportamiento caótico, pequeñas variaciones en las configuraciones iniciales pueden provocar evoluciones diferentes.
- **Tipo 4**: Generalmente se engloban en este tipo aquellos autómatas célulares que pasan por una larga fase evolutiva antes de caer en un atractor.

2.2.4. Modelado de un sistema

En ingeniería los procesos y sistemas son generalmente complicados, por lo que son simplificados mediante idealización y aproximaciones para poder resolver el problema planteado. El proceso de simplificación de un problema, para poder ser representado en términos de un sistema de ecuaciones para el análisis, diseño y optimización, o a través de un arreglo físico para la experimentación, se denomina modelado.

Los modelos se dividen en dos géneros: descriptivos y predictivos. Los primeros se usan para explicar los mecanismos y principios que interactúan en un sistema. Los últimos son los que se emplean en el diseño y optimización de sistemas en ingeniería; estos son capaces de predecir el comportamiento de un sistema dado y se clasifican en cuatro tipos: modelos análogos, modelos matemáticos, modelos físicos y modelos numéricos (Edgar y Himmelblau, 1988).

Los modelos análogos están basados en fenómenos físicos existentes; estos permiten el uso de la solución de un problema relacionado para obtener los correspondientes resultados de algunos problemas que no han sido resueltos. El modelo matemático, en contraste, representa el desempeño y comportamiento de un sistema dado en términos de ecuaciones matemáticas, ofreciendo resultados cuantitativos. El modelo físico representa un sistema, el cual se usa generalmente para obtener resultados experimentales sobre el comportamiento del mismo. Los modelos numéricos están basados en los modelos matemáticos y permiten obtener el comportamiento del sistema para diferentes condiciones de operación y diferentes parámetros de diseño. Un aspecto importante es que muy pocos problemas pueden ser resueltos por procedimientos analíticos, siendo

necesario el uso de métodos numéricos para resolver las ecuaciones que representan los sistemas reales. Los métodos numéricos requieren de los recursos computacionales para resolver las ecuaciones del sistema.

2.2.5. Optimización

En el mundo real la mayoría de los problemas tienen varias soluciones y algunos de ellos tienen soluciones infinitas. En el área de optimización, el propósito principal es identificar o encontrar la mejor solución posible, entre el conjunto de todas las soluciones potenciales para un problema dado. Para encontrarla, se realiza una selección de valores para un número de variables interrelacionadas, mejor conocidas como variables de diseño o de decisión; se requiere de una o varias funciones objetivo, las responsables de modelar el sistema analíticamente. Esta función o funciones objetivo adquieren valores para las diferentes variables de decisión. Los valores sirven para verificar la calidad de las variables de decisión. Esta función o funciones objetivo son maximizadas o minimizadas, dependiendo de la formulación del problema; y pueden o no, estar sujetas a restricciones que limitan la selección de valores de las variables.

Un problema de optimización se puede describir formalmente como un par (F, c), donde F es el dominio de los puntos factibles y c es la función de costo, $c : F \to \mathbb{R}$. El problema consiste en encontrar $f \in F$ tal que: $c(f) \leqslant c(y)$, $\forall y \in F$ (problema de minimización). Dicho punto se denomina solución globalmente óptima de un caso dado. Un problema de optimización es un conjunto de casos, donde un caso es equivalente a los datos de entrada y un problema a una colección de casos.

2.2.5.1. Optimización combinatoria

Existe una clase especial de problemas que son tratados mediante optimización combinatoria. Este tipo de optimización es un área de las matemáticas aplicadas donde se combina técnicas de combinatoria, programación lineal y teoría de algoritmos para resolver problemas de optimización sobre estructuras discretas como enteros, permutaciones

y grafos. Los problemas de optimización combinatoria aparecen en las áreas de manufactura, transportación, telecomunicaciones, computación, biología computacional, finanzas, etcétera. La mayoría de estos problemas son de extrema complejidad y no pueden ser resueltos mediante técnicas convencionales de optimización (Nemhauser y Wolsey, 1988).

2.2.5.2. Problemas matemáticos de optimización

Un problema matemático de optimización consiste en la maximización o minimización de funciones algebraicas de una o más variables (Cuthbert, 1987). La selección de los valores de las variables pueden estar restringidas por ecuaciones o desigualdades algebraicas llamadas restricciones, de tal forma que el objetivo no es encontrar el mejor valor posible sino el mejor valor permitido por las restricciones. Un ejemplo es el siguiente, encontrar x tal que maximice o minimice $f(x)$, sujeto a las restricciones $g_j(x) = b_j\{j = 1, \ldots, m\}$ y $g_k(x) \leq c_k\{k = 1, \ldots, p\}$, donde $x \in \Re^n$ (vector de n componentes). La rama de las matemáticas aplicadas que resuelve problemas de optimización se denomina “programación matemática”, donde $f(x)$ es la función objetivo y una solución posible al problema anteriormente formulado es el mejor valor permitido por $g_j(x)$ y $g_k(x)$. La optimización computacional es una herramienta fundamental en los procesos de toma de decisión. En la Figura 7 se muestra la clasificación de los métodos de optimización, usando programación matemática.

La programación lineal maneja problemas donde la función objetivo y las restricciones son combinaciones lineales de las variables de decisión. Para la programación no-lineal se utiliza un conjunto de técnicas para optimizar funciones no-lineales sujetas a restricciones de igualdad o desigualdad; tanto las funciones como las restricciones pueden ser de una o más variables. La mayoría de los algoritmos en programación no-lineal son iterativos, mientras que en programación lineal existe una secuencia de longitud finita para alcanzar la solución. En programación no-lineal la secuencia generalmente no alcanza la solución óptima, sino que converge hacia ella, es decir, en problemas no-lineales se determina una solución lo suficientemente cercana a la óptima.

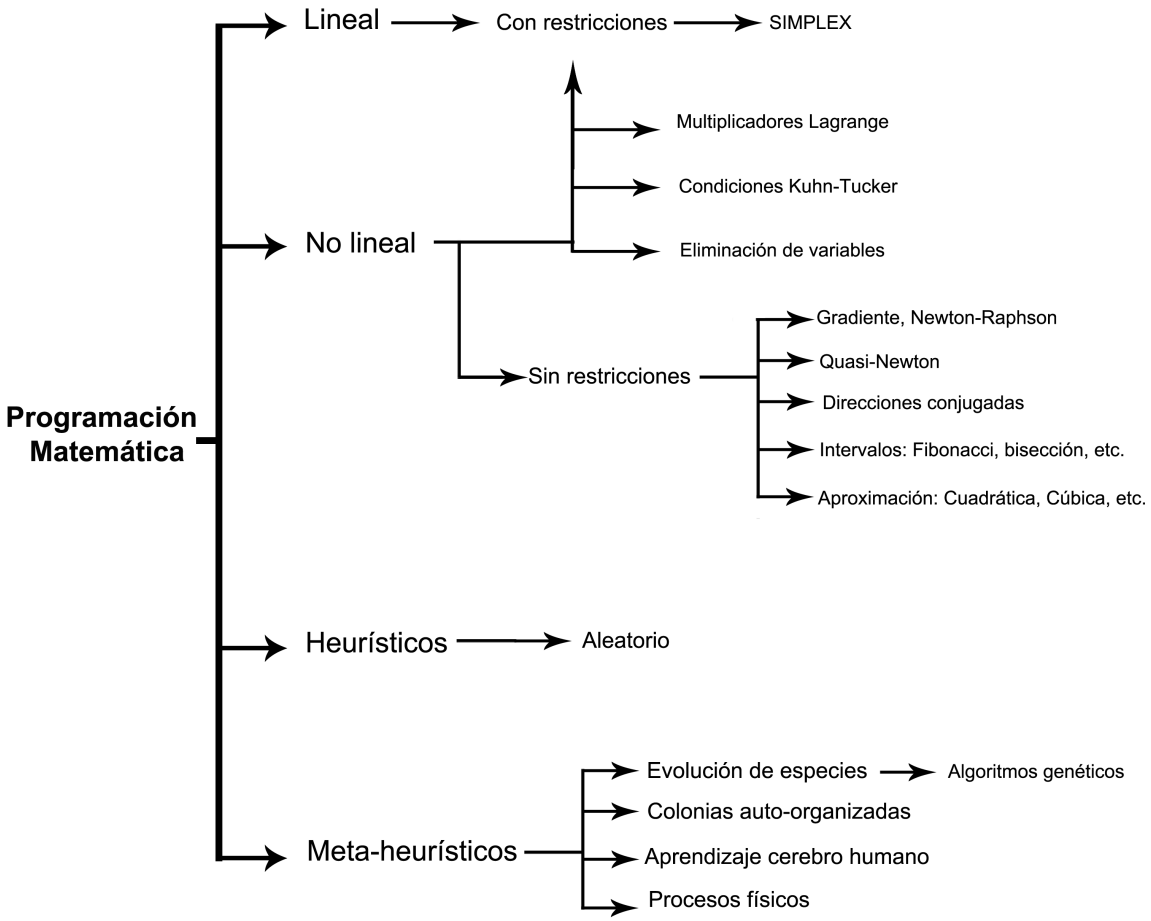


Figura 7: Clasificación de métodos de optimización.

Los métodos heurísticos son aleatorios y se basan en la generación de una secuencia de aproximaciones mejoradas, cada una de las cuales se deriva de la aproximación previa. Por ejemplo, si x_i es la aproximación obtenida en la iteración i, entonces la nueva aproximación en la etapa $i+1$ se obtiene $x_{i+1} = x_i + \lambda \cdot u_i$, donde λ es un longitud de paso (valor escalar) y u_i es un vector aleatorio unitario generado en la i-ésima iteración. Si $f(x_{i+1}) > f(x_i)$, entonces x_{i+1} se acepta como el nuevo punto.

Las meta-heurísticas generalmente se aplican a problemas que no tienen un algoritmo o heurística específica o bien, cuando no es posible implementar un método óptimo. Entre las más conocidas se encuentran: evolución de especies (algoritmos genéticos), colonia de hormigas, aprendizaje cerebro humano y procesos físicos.

2.2.6. Computación evolutiva

El paradigma de computación evolutiva está inspirado en la evolución de especies (teoría darwiniana). La teoría de la evolución describe como las especies naturales evolucionan para adaptarse a un entorno y aquellos individuos que tengan éxito en tal adaptación tendrán mayor probabilidad de sobrevivir hasta la edad adulta y probablemente un número mayor de descendientes, por lo tanto, sus genes tienen alta probabilidad de ser propagados a lo largo de generaciones sucesivas. La combinación de características de los padres bien adaptados, en un descendiente, puede producir muchas veces un nuevo individuo mucho mejor adaptado que cualquiera de sus padres a las características de su medio ambiente.

Este proceso no debe verse en ningún momento como un proceso determinista, sino como un proceso con una fuerte componente estocástica. Es decir, si un individuo se adapta al entorno, lo más que se puede afirmar es que ese individuo tendrá mayor probabilidad de conservar sus genes en la siguiente generación que sus congéneres. Pero sólo es probable, no es un evento con certeza absolutamente. Siempre existirá la posibilidad de que a pesar de estar muy dotado por alguna razón no consiga reproducirse. Pero en cuanto a la especie como un conjunto o población, sí puede afirmarse que ésta irá adaptándose al medio (Seckbach, 2012).

La idea la propuso Holland quien, consciente de la importancia de la selección natural, introdujo la idea de los Algoritmos Genéticos en los años 70s y al final de esa década desarrolló una técnica que permitió incorporarla en un programa de computadora. Su principal objetivo era lograr que las computadoras aprendieran por sí mismas. A la técnica inventada por Holland se le llamó inicialmente “Planes reproductivos”, pero se hizo popular bajo el nombre de Algoritmos Genéticos (AG).

Los algoritmos genéticos son una de las más conocidas y originales técnicas de resolución de problemas dentro de lo que se ha definido como “Computación evolutiva” (o “Algoritmos evolutivos”), término que agrupa a los AG, las estrategias evolutivas y la programación evolutiva. En realidad todas estas técnicas son muy parecidas y comparten muchos aspectos.

2.2.6.1. Algoritmos genéticos

Los algoritmos genéticos son una opción interesante para resolver problemas de optimización. Un algoritmo genético es un método de búsqueda dirigida basada en probabilidad. Bajo la condición de selección de individuos, el algoritmo converge en probabilidad al óptimo, es decir, al aumentar el número de iteraciones la probabilidad de tener el óptimo dentro de la población tiende a uno (Eiben y Smith, 2003).

La idea inicial parte de, dada una población de individuos en un ambiente con recursos limitados, compiten por estos recursos mediante selección natural o supervivencia del más apto. Evolucionan una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica como cruce y mutación genética, así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados (Chambers, 1998). Generalmente este criterio es una función que describe un problema de optimización.

La representación del individuo esta altamente relacionada con el problema a tratar. Los resultados arrojados por el algoritmo genético dependen del diseño de la representación del individuo, por ello su importancia. Para generar la población inicial, es importante que la aleatoriedad sea alta; entre más distintos sean los individuos en la población, mayor probabilidad habrá de encontrar un individuo que represente una solución optima.

El funcionamiento de un algoritmo genético se ilustra en la Figura 8, donde se parte de la población inicial previamente generada, para proseguir a la selección de los padres. Se seleccionan al azar dos padres de la población de individuos; a los cuales posteriormente se les aplica el operador de cruce o recombinación. La función que desempeña el operador de cruce es mezclar las características de los individuos padres y producir hijos con aptitudes diferentes a ellos. Consecuentemente, a los individuos cruzados se les aplica otro operador genético: la mutación. Esta tiene como objetivo realizar un pequeño cambio en un individuo, generando un individuo mutado, para mantener la diversidad genética en la población. Como en la naturaleza, la mutación debe ser un cambio mínimo en el indivi-

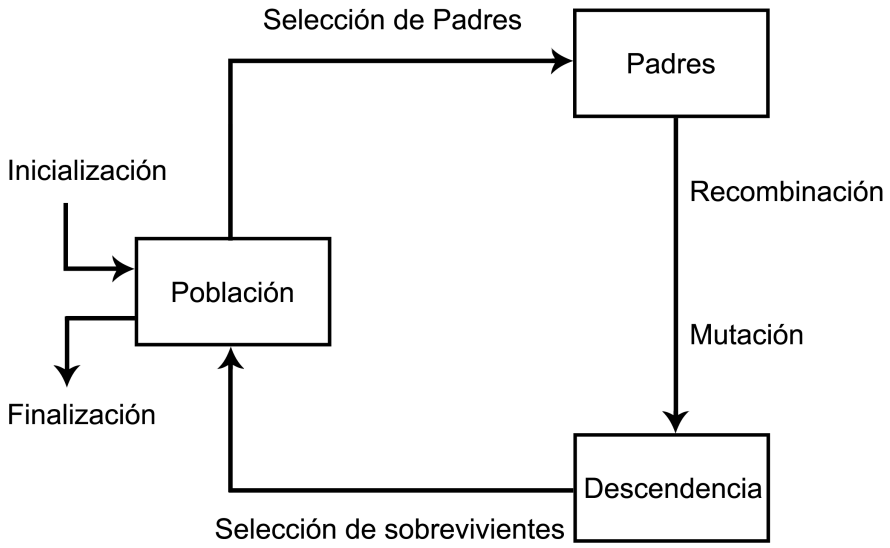


Figura 8: Esquema general de un algoritmo genético.

duo. Posteriormente, habiendo sido aplicados ambos operadores genéticos, se procede con la evaluación de los individuos con base en su función objetivo (maximizar o minimizar). Esto es, los individuos con aptitud mayor son seleccionados para formar parte de la población de la siguiente generación. Podemos observar que el funcionamiento de un algoritmo genético es un ciclo, el cual tiene como condición de paro dos aspectos: uno, que en la población ya no haya un cambio significativo entre los individuos, en otras palabras, que la población converja a un individuo suficientemente apto. Dos, una cantidad de generaciones previamente definida.

El diseño del algoritmo genético para el problema PATS se describe detalladamente en el Capítulo 3.

2.3. Cómputo biomolecular

El área de cómputo biomolecular es muy extensa, por ello en esta sección se presenta una descripción breve de la misma. El cómputo biomolecular o cómputo con ADN es un paradigma de cómputo no convencional, utiliza las diversas propiedades de las móleculas biológicas para realizar cómputo. Las ventajas de la computación por ADN se basan en dos características fundamentales:

La primer característica es el gran paralelismo de las hebras de ADN. Muchos de los problemas considerados intratables, pueden ser resueltos haciendo una búsqueda exhaustiva sobre todas las soluciones posibles. Sin embargo, a la fecha, la dificultad consiste en el hecho que tal búsqueda es demasiado grande como para poder ser realizada usando la tecnología actual. Por otro lado, la densidad de información almacenada en hebras de ADN y la facilidad de construir muchas copias de ellas podrían convertir esas búsquedas en una posibilidad real.

La segunda es la complementariedad de Watson-Crick. La complementariedad es algo que viene sin costo en la naturaleza. Cuando se unen dos hebras (en condiciones ideales) se conoce el opuesto a cada miembro, no hay necesidad de verificarlo.

2.3.1. El experimento de Adleman

En 1994, Adleman publicó un artículo describiendo un experimento que utilizaba el ADN como un sistema computacional. En él resuelve una instancia con siete nodos del problema del camino Hamiltoniano, utilizando moléculas de ADN y técnicas de laboratorio de biología molecular para realizar cómputo con ellas (Adleman, 1994). Adleman fue el primero en lograr resultados experimentales, aunque su solución resultó trivial, dado que fue un caso pequeño y particular. Dado un grafo dirigido $G = (V, E)$ con $V = \{v_1, v_2, ..., v_n\}$ y un par de vértices v_{i_1} y v_{i_n} designados del grafo, una trayectoria de Hamilton consiste en una secuencia $\langle v_{i_1}, v_{i_2}, ..., v_{i_{n-1}}, v_{i_n} \rangle$ de los n vértices de manera que $(v_{i_r}, v_{i_{r+1}}) \in E, \forall (1 \leqslant r < n)$.

Cada paso del Algoritmo 1 es un procedimiento en laboratorio biomolecular. Para la codificación del grafo se diseñaron secuencias de 20 nt en el paso 1. La secuencia de cada arista dirigida se divide en dos partes, cada una corresponde a una mitad del complemento de la secuencia de un vértice origen v_0 y un vértice destino v_d. Así, las secuencias que codifican vértices se unen a secuencias que codifican aristas para posteriormente ligarse entre sí. En el paso 2, se amplifican mediante PCR las hebras obtenidas en el paso 1, utilizando como *primers* las secuencias correspondientes a v_{i_1} y el complemento de v_{i_n}.

Algoritmo 1 Rutas de Hamilton- Adleman

Entrada: Grafo $G = (V, E)$ con $V = \{v_1, v_2, ..., v_n\}$, vértice inicial v_{i_1} y final v_{i_n}.

Salida: Trayectoria(s) de Hamilton en G.

1: Generar trayectorias aleatorias en G.
2: Mantener sólo aquellas trayectorias que comiencen en v_{i_1} y finalicen en v_{i_n}.
3: Mantener sólo aquellas trayectorias que contengan exactamente n vértices.
4: Mantener sólo aquellas trayectorias que consideren cada vértice al menos una vez.
5: Leer las trayectorias resultantes (si existen).

Sólo las hebras que inician en v_{i_1} y finalizan en v_{i_n} son amplificadas. El paso 3 se realiza separando las hebras por tamaño a través de electroforesis en gel. Las hebras que incluyen exactamente los 7 vértices del caso particular tienen una longitud de $7 \times 20 = 140$ nt. En el paso 4, para cada vértice se aplica el procedimiento de purificación por afinidad de forma iterativa. En cada iteración $1 \leqslant i \leqslant n$ se mantienen las hebras que incluyen al *i*-ésimo vértice. Finalmente, para el paso 5 se amplifican las hebras, en caso de existir, para su detección.

Para realizar el experimento reportan que se paralelizó masivamente 1023 moléculas diferentes de ADN para encontrar una solución simultáneamente; los experimentos tuvieron una duración de menos de una semana (Adleman, 1994). Se utilizó aproximadamente 10^{-10} % del uso de energía de una supercomputadora. En términos de tiempo de ejecución, el número de operaciones para verificar que una molécula sea un camino Hamiltoniano es lineal al número de vértices $(O(n))$.

El experimento de Adleman se considera como base para otros experimentos, sin embargo, no se puede considerar como un esquema algorítmico molecular, dado que el tubo de ensayo inicial del experimento sólo se puede utilizar para el grafo concreto para el que fue diseñado; en caso de considerar otro grafo, se requiere la elaboración nuevamente del tubo de ensayo inicial. No obstante, la formulación abstracta del algoritmo de Adleman permite elaborar una primer plantilla de programación molecular para resolver el problema del camino Hamiltoniano.

2.3.2. Modelos de cómputo con ADN

Los modelos en computación basada en ADN se clasifican como: modelos clásicos y modelos autónomos. A continuación se enlista cada clase con sus respectivos modelos.

Modelos clásicos

- Modelo de filtrado
- Modelo de concatenación
- Modelo de etiquetas

Modelos autónomos

- Auto-ensamblamiento de ADN
- Computación con estructuras secundarias de ADN
- Autómatas moleculares

En este trabajo utilizaremos el modelo de auto-ensamblado de ADN, por lo cual, se describirá detalladamente.

2.3.3. Auto-ensamblado de ADN

El problema de auto-ensamblado de mosaicos (*The tiling Problem*) fue introducido por (Wang, 1960) para estudiar una pregunta de lógica matemática. La pregunta es simple: dado un conjunto finito de mosaicos geométricos (por ejemplo polígonos), determinar cuántos mosaicos son necesarios para cubrir un plano completo sin superponerse. La respuesta a esta simple pregunta es que no existe ningún algoritmo que otorgue la respuesta correcta para todos los casos (Berger, 1966). La demostración de este resultado se basa en la construcción de patrones de ensamblamiento capaces de simular una máquina de Turing.

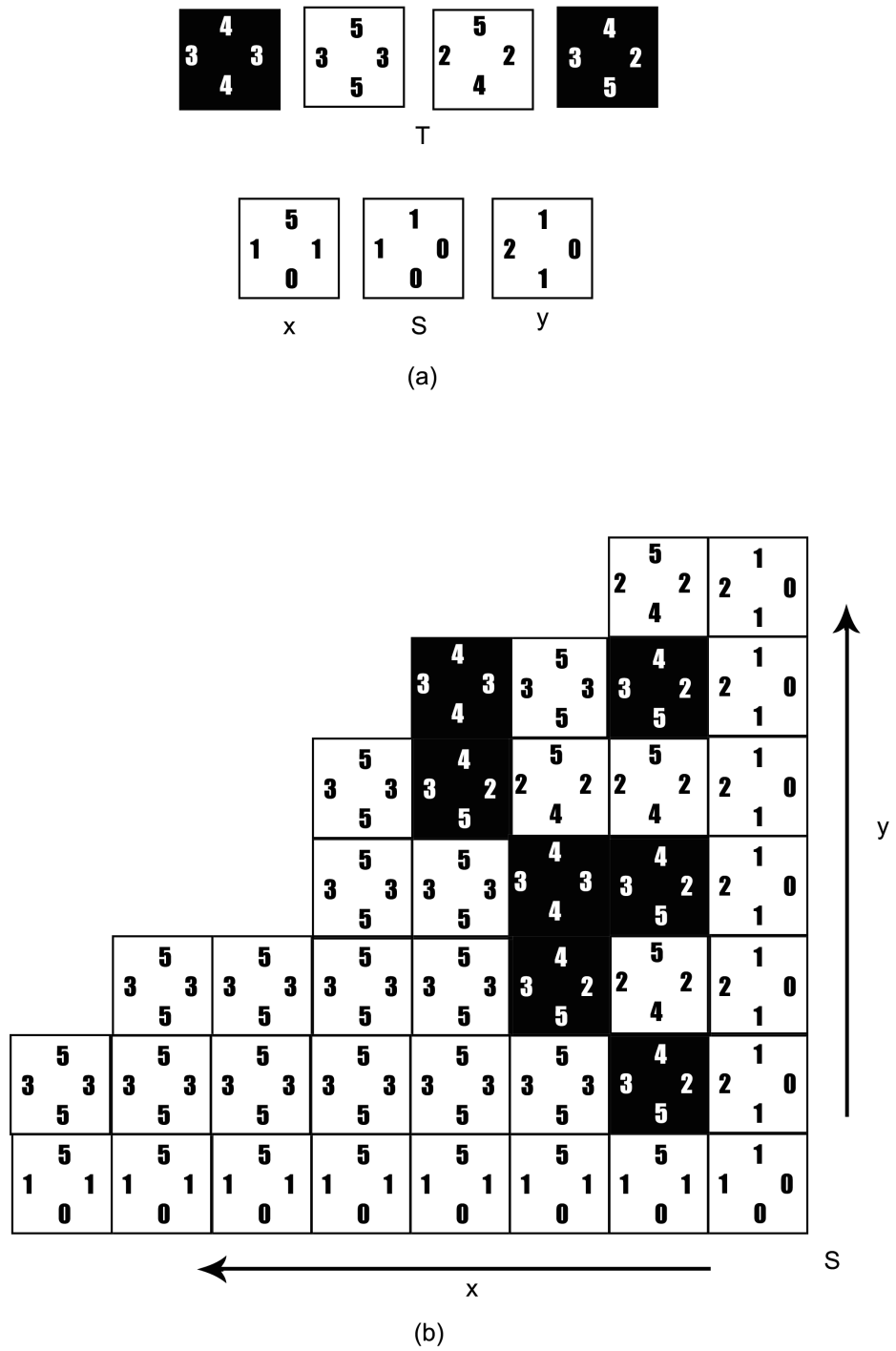


Figura 9: Un sistema de ensamblaje para el patrón Contador Binario: (a) conjunto T con cuatro mosaicos, mosaico semilla S y mosaicos de acotamiento x y y, (b) ensamble de mosaicos que forman el patrón.

Un ejemplo de cómo se auto-ensamblan los mosaicos se ilustra en la Figura 9, en el que un conjunto de 7 mosaicos se ensamblan al mismo tiempo para formar un contador binario. La forma de representar la información en los mosaicos determina el cómputo que

se realiza mediante el ensamblaje. Los lados superior e inferior del mosaico codifican el valor del bit en el contador, mientras que los lados de la derecha y de la izquierda se usan para indicar si el bit debe o no cambiar su valor. El mosaico semilla S activa el cómputo y los 2 mosaicos de acotamiento proveen las condiciones iniciales. Los mosaicos de acotamiento por abajo codifican los bits iniciales (000). Los mosaicos de acotamiento por la derecha producen una serie de "comandos" para incrementar el último bit en cada columna. Por último, los cuatro mosaicos que representan la regla de cálculo, implementan la siguiente lógica: si el bit de la derecha no es de cambio, el bit mantiene su valor; si el bit de la derecha es de cambio, 0 cambia a 1, y 1 cambia a 0.

Es importante notar que, para que en el ejemplo anterior el patrón resultante sea el correcto, es necesario que los mosaicos de acotamiento se ensamblen al mosaico semilla antes de usar los mosaicos de regla y que los mosaicos de regla sean agregados solamente cuando ya están presentes los mosaicos de la derecha y los de abajo. Estas condiciones previenen la formación de errores de ensamblado por ausencia de mosaicos de acotamiento, como también previenen la formación de ensambles incompletos. Eric Winfree mostró cómo, usando la temperatura, se puede implementar el auto-ensamblado con cierto tipo de "mosaicos de ADN" construidos con secuencias específicas de hibridación y diseñadas para "auto-ensamblarse" espontáneamente y formar ensambles que realicen cómputos o adopten una forma o figura predefinida (Winfree *et al.*, 1998).

2.3.3.1. aTAM (*abstract Tile Assembly Model*)

Formalmente, un mosaico se puede definir como $t = (\sigma_N, \sigma_E, \sigma_S, \sigma_O)$, $\sigma_i \in \Sigma$, es una unidad cuadrada y orientada, con sus lados norte, este, sur y oeste etiquetados con un determinado pegamento de un alfabeto Σ. Sea $g : \Sigma \times \Sigma \to \mathbb{N} \bigcup \{0\}$, si $g(x, y) = g(y, x)\ \forall\ x, y \in \Sigma$ y $g(null, x) = 0\ \ \forall\, x, y \in \Sigma$, entonces $null \in \Sigma$. (Winfree *et al.*, 1998).

Para cada mosaico t, denotamos la etiqueta de su lado $i \in \Delta = \{N, E, S, O\}$ como $\sigma_i(t) \in \Sigma$. La representación de la dirección opuesta de i es i^{-1}. Además se usan las orientaciones para representar funciones de tipo $\mathbb{Z}^2 \to \mathbb{Z}^2$: N(x, y) = (x, $y + 1$), S(x, y)= (x, $y - 1$), E(x, y)=($x + 1$, y), O(x, y)=($x - 1$, y). El par ordenado (x, y) representa la coordenada donde se ubica el mosaico en el ensamble.

Sea T un conjunto finito de mosaicos. Suponemos que T contiene un mosaico especial ***vacío*** = $(null, null, null, null) = \varnothing$. Una configuración de T es una función $C : \mathbb{Z} \times \mathbb{Z} \to T$. Se escribe como $(x, y) \in C$ si y sólo si $C(x, y) \neq$ ***vacía***. Una configuración C se dice no vacía si existe $(x, y) \in C$. Dado un mosaico $t \in T$, y un par $(x, y) \in \mathbb{Z} \times \mathbb{Z}$, denotado por:

$$C_t^{(x,y)}(i,j) = \begin{cases} t & \text{si } (i,j) = (x,y), \\ \text{vacío.} & \end{cases} \tag{2}$$

Se define la ***energía*** G de una configuración C como la suma de todas las fuerzas de interacción entre mosaicos. La función g determina las fuerzas de pegado en los mosaicos.

$$G(C) = \frac{1}{2} \sum_{x,y \in Z} \sum_{i \in \Delta} g(\sigma_i(C(x,y), \sigma_{i^{-1}}(C(i(x,y))).$$

Una configuración C es un $\tau - ensamble$ si no puede ser particionado en dos subconfiguraciones no vacías C_1 y C_2 tal que la fuerza de pegado en la frontera entre ellas sea menor que la temperatura τ, es decir, $g(C_1, C_2) < \tau$ donde $g(C_1, C_2) = G(C) - [G(C_1) + G(C_2)]$.

Un sistema de mosaicos T es una 4-tupla $\langle T, s, g, \tau \rangle$ donde T es el conjunto de mosaicos, s es un mosaico especial llamado *semilla*, g es la función de fuerzas de pegado y $\tau \geqslant 0$ es la temperatura. Un mosaico t puede ser agregado a un $\tau - ensamble$ *C* si la suma de las fuerzas de pegado entre t y sus vecinos en *C*, es al menos la temperatura τ.

Complejidad de producir un $\tau - ensamble$

Se define la *complejidad de programación* de un ensamble específico A como el número mínimo de tipos de mosaicos distintos que se requieren para producir únicamente A.

En (Rothemund y Winfree, 2000) se muestra la complejidad en la programación de un cuadrado de $n \times n$ cuando $\tau = 1$ en n^2. En general, si definimos el tamaño de un ensamble como el número de nodos, entonces la complejidad de programación de cualquier

ensamble es del orden de su tamaño. Para $\tau \geq 2$, Rothemund y Winfree (2000) prueban que la complejidad en la programación del cuadro de $n \times n$ es $\Theta(log(n)/log(log(n)))$. En su construcción utilizan $\tau = 2$.

Adleman y su equipo de trabajo demuestran que el problema de determinar un sistema de mosaicos minimal que produzca únicamente un ensamblado específico es NP-completo, aunque para ciertas familias de ensamblajes, pueden ser resueltos en tiempo polinomial (Adleman *et al.*, 2002).

2.3.4. Herramientas computacionales

Dentro de la literatura, existen dos herramientas para realizar la simulación del autoensamble de mosaicos: Xgrow e ISU TAS.

2.3.4.1. Xgrow

El simulador Xgrow está escrito en C, fue desarrollado en el Instituto Tecnologico de California (Caltech), por un grupo de investigadores de varias instituciones. Es una creación de trabajo colaborativo y se ha utilizado para simular ensamblados en diferentes trabajos de investigación.

Xgrow implementa el "*abstract Tile Assembly Model*" (aTAM) y el "*kinetic Tile Assembly Model*" (kTAM). Estos modelos consideran un solo cristal que crece en un entorno de solución por los mosaicos de monómeros individuales, uno a la vez. En el kTAM, los mosaicos individuales también pueden caerse. Los parámetros de entrada para una simulación en Xgrow son los siguientes:

- un archivo que contiene las definiciones para todo tipo de mosaicos que se van a utilizar especificando las fuerzas de pegado en los lados del mosaico y color.
- el parámetro Gmc, que exponencialmente dicta la concentración de los mosaicos.
- el parámetro Gse, es la fuerza de un vínculo "unidad" y por lo tanto dicta exponencialmente la velocidad de disociación.

Dependiendo de los parámetros, el simulador hace crecer lentamente el ensamblaje (justo en equilibrio), o crecer rápidamente y sin control (cuando las tasas son demasiado altas). Los modelos básicos aTAM y kTAM suponen que los mosaicos de monómeros están presentes a una concentración fija durante todo el ensamblaje. Xgrow puede simular muchos copos simultáneamente. Esto es interesante en el caso donde el crecimiento de cristales agota la concentración de monómeros.

La característica principal de Xgrow es que, por lo general es rápido. Todos los posibles eventos en el proceso de Markov se almacenan en un árbol cuádruple para la selección en tiempo logarítmico para el siguiente evento. La desventaja es que no es gráfico.

2.3.4.2. ISU TAS

ISU TAS es un simulador que usa el "*abstract Tile Assembly Model*" (aTAM). Este novedoso simulador permite a los usuarios diseñar mosaicos y semillas con el fin de simular el ensamblado entre ellas.

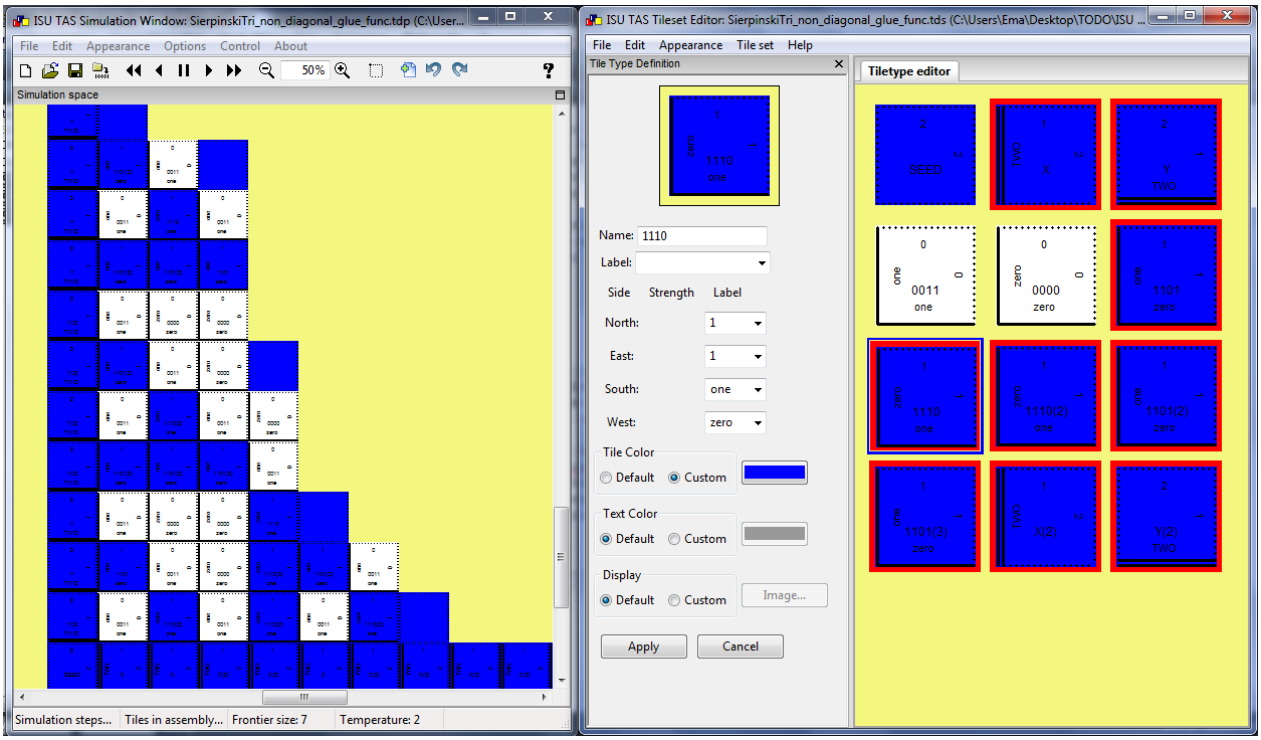


Figura 10: Interfaz de ISU TAS

El simulador permite la creación gráfica de los ensambles partiendo de la semilla (Figura 10), el avance rápido y el rebobinado de crecimiento del ensamble. Es intuitivo y fácil de usar, cuenta con distintas funcionalidades tales como realizar acercamientos al ensamble, desplazamientos, y la inspección de los ensambles, entre otras características. El editor gráfico del tipo de mosaicos permite manipular fácilmente el diseño de mosaicos. Ensamblajes y conjuntos de mosaicos se pueden crear, guardar y volver a cargar. ISU TAS es liberado bajo la licencia GPL y código fuente, esto lo hace accesible para el público en general.

```
TILENAME SEED
LABEL
NORTHBIND 0
EASTBIND 0
SOUTHBIND 0
WESTBIND 0
NORTHLABEL 2
EASTLABEL 2
SOUTHLABEL
WESTLABEL
TILECOLOR blue
CREATE
```

```
TILENAME X
LABEL
NORTHBIND 0
EASTBIND 0
SOUTHBIND 0
WESTBIND 2
NORTHLABEL 1
EASTLABEL 2
SOUTHLABEL
WESTLABEL TWO
TILECOLOR blue
CREATE
```

```
TILENAME Y
LABEL
NORTHBIND 0
EASTBIND 0
SOUTHBIND 2
WESTBIND 0
NORTHLABEL 2
EASTLABEL 1
SOUTHLABEL TWO
WESTLABEL
TILECOLOR blue
CREATE
```

Figura 11: Formato ISU TAS para el diseño de mosaicos.

El formato que utiliza ISU TAS para el diseño de mosaicos se muestra en la Figura 11. El mosaico cuenta con características como fuerza de pegado para cada orientación del mosaico y adicionalmente utiliza etiquetas para cada fuerza de pegado, también tiene color y nombre. El nombre determina si el mosaico es parte de la semilla o no. El diseño de tres mosaico para para formar un patrón de Sierpinski se muestra en la Figura 11 y el auto-ensamble de los mismos se presenta en la Figura 10.

2.3.5. Algoritmos que abordan el problema PATS

En esta sección se describen los tres algoritmos que abordan al problema PATS y que constituyen el estado del arte del problema.

Inicialmente, Ma y Lombardi (2008) diseñan los algoritmos PATS_Tile y PATS_Bond para solucionar el problema PATS, los cuales utilizan una heurística de búsqueda voraz.

Para la representación del problema se utiliza un estructura de grafo. En ambos algoritmos los vértices del grafo corresponden a los mosaicos de ADN del sistema y las aristas a los extremos pegajosos, sin embargo, la diferencia radica en la manera en que se les asigna el color. En el algoritmo PATS_Tile colorea los mosaicos de ADN, mientras que en PATS_Bond colorea las aristas. Ellos, como pioneros del problema PATS, realizan la simulación de los algoritmos antes mencionados utilizando casos de prueba que forman patrones tales como el triángulo de Sierpinski (con instancias de 6×6, 8×8, 12×12, 17×17 y 32×32), el contador binario (con instancias 6×6 y 15×15) y el tablero de ajedrez (con instancias 6×6 y 15×15).

Los resultados fueron comparados con el denominado conjunto trivial de mosaicos. Este conjunto se define trivial, dado que todos los mosaicos en el conjunto son de diferente tipo, donde el tipo se define como la combinación de las fuerzas de pegado del mosaico con el color del mismo. Las simulaciones arrojan resultados de 41 % al 47 % de reducción con respecto al conjunto trivial de mosaicos (Ma y Lombardi, 2008). Sin embargo, estos resultados son únicamente el punto de referencia para futuros trabajos.

En (Göös y Orponen, 2010) se presenta un algoritmo de búsqueda exhaustiva "*branch-and-bound*" (B&B). Ellos extienden el método de Ma y Lombardi (2008) para obtener el algoritmo antes mencionado. En lugar de usar una búsqueda donde se elige la mejor solución en cada paso, como en el trabajo original, se realiza una búsqueda a profundidad donde se va sesgando el campo de soluciones. Los resultados arrojados por la simulación son notables, por ejemplo, con el patrón de Sierpinski de 8×8 se obtuvo un 58 % de reducción en el conjunto de mosaicos, mientras que en (Ma y Lombardi, 2008) fue de un 43 %. Mencionan que para obtener el 58 % de reducción, detuvieron el algoritmo en el paso 10^6, por lo que dejan abierto el camino a una solución con un porcentaje de reducción mayor. En el caso del patrón de Sierpinski de 12×12 se obtuvo un 90 % de reducción, mientras que Ma y Lombardi (2008) presentaron un 45 %. Concluyen que el algoritmo B&B, propuesto en su trabajo, obtiene el óptimo para patrones de tamaños de hasta 6×6, y soluciones aproximadas para patrones más grandes.

En el trabajo de Lempiäinen *et al.* (2011) modifican el algoritmo B&B para hacerlo más eficiente utilizando una combinación de estrategias "*Partition-Search*" (PS) con "*Branch-*

and-Bound" (B&B), y mediante la ejecución de varias instancias en paralelo, acortan el tiempo de búsqueda considerablemente. Entre los resultados obtenidos, destaca que en el patrón de Sierpinski de 32×32 se alcanzo el 37 % de reducción en tan solo 30 segundos y en el de Sierpinski de 64×64 con una reducción de 34 % en una hora. También presentan un método para calcular la fiabilidad de un conjunto de mosaicos dado, es decir, la probabilidad de errores de pegado en el proceso del auto-ensamblaje, basado en el análisis de Winfree (1998). Presentan los datos empíricos sobre la fiabilidad de los conjuntos de mosaicos encontrados por el algoritmo de PS-BB.

En este capítulo se presentaron conceptos básicos que constituyen el marco teórico de esta investigación, en el próximo capítulo (Capítulo 3) se describe detalladamente el diseño del algoritmo genético para el problema PATS.

Capítulo 3. Diseño de un algoritmo genético para el problema PATS

En el modelo de auto-ensamblado se tratan diversos problemas, entre ellos problemas de optimización. El problema "*Patterned self-Assembly Tile set Synthesis*" (PATS), como se menciona en el primer capítulo, es un problema de optimización. Su función objetivo es minimizar el número de tipos de mosaicos en el conjunto. A este problema, se le han aplicado diferentes algoritmos de búsqueda determinísticos. Sin embargo, el problema PATS tiene un espacio de soluciones extenso dado que es un problema NP-difícil (Czeizler y Popa, 2012). La aplicación de un algoritmo genético sería viable, principalmente, por la naturaleza de búsqueda que realiza. Cabe resaltar, que en la literatura no se ha aplicado este tipo de técnicas al problema.

A continuación se describe el diseño de un algoritmo genético para el problema PATS (AG-PATS) de auto-ensamblado de ADN.

3.1. Problema de optimización

El problema PATS, se puede definir como:

Dado: un patrón bidimensional P, donde $P = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$ tal que

$$a_{i,j} = \begin{cases} 1 & \text{si } el\ color\ es\ negro, \\ 0 & \text{si } el\ color\ es\ blanco. \end{cases} \tag{3}$$

Encontrar: un sistema de mosaicos **T** $= \langle T, S, g \rangle$ tal que

- El conjunto T de tipos de mosaicos sea de cardinalidad mínima.
- El sistema de mosaicos **T** auto-ensamble el patrón P.

3.1.1. Función objetivo

La ecuación 4 muestra los elementos involucrados en la función objetivo del problema PATS, donde k son los k-colores y m_i son los tipos de mosaicos. No obstante, el problema utiliza únicamente dos colores (blanco y negro) por lo que $k = 2$.

$$\min \sum_{i=1}^{k} \sum_{j=1}^{m_i} t_{i,j} =\mid T \mid . \tag{4}$$

El color de los mosaicos (blanco y negro) es únicamente representativo, esto es que físicamente las moléculas DAE y DAO no tienen un color. Cabe mencionar que la molécula DAO está compuesta por 37 nucleótidos y la molécula DAE por 42 nucleótidos, por lo que en el ensamblaje obtenido del auto-ensamblado de estas moléculas se visualiza (mediante microscopio) de dos colores. En el ensamblaje se percibe un relieve que provoca un cambio de color entre las moléculas ensambladas y por ello se abstraen en forma de dos colores, el relieve de color blanco y el otro negro (Rothemund *et al.*, 2004). El tipo de mosaico se define como la combinación de color con fuerzas de pegado en el mosaico. El objetivo de la función es minimizar el tipo de mosaicos en el conjunto de entrada, necesarios para ensamblar el patrón P deseado (Ma y Lombardi, 2008).

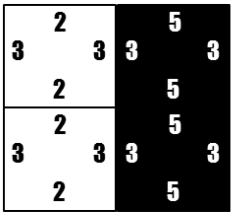
3.2. Representación del individuo

La representación del individuo es parte fundamental para el diseño del algoritmo genético, ya que está intimamente ligada a la calidad de las soluciones obtenidas (Eiben y Smith, 2003). En otras palabras, con una representación adecuada del individuo, resaltando los aspectos a evolucionar, se podrá obtener buenas soluciones.

Para el problema PATS se propone la representación del individuo como una matriz de mosaicos que forman el patrón P dado. Sin embargo, podríamos cuestionarnos, ¿por qué una matriz y no un grafo? Inicialmente el problema PATS fue introducido con una representación de grafo (Ma y Lombardi, 2008). No obstante, la decisión de representarlo como una matriz, surgió bajo el principio de visualizar al individuo como una imagen

formada de pixeles, donde cada pixel representa un mosaico, y cada mosaico tiene su lugar en la imagen que forma el patrón.

Entonces, la estructura del individuo es una matriz de mosaicos, los cuales contienen un conjunto de características (color, fuerzas de pegado en orientación norte, sur, este y oeste). En algoritmos genéticos, los individuos son concebidos como cromosomas que contienen las características del individuo que se desea evolucionar. La validez del individuo se determina mediante dos restricciones. Primero, no debe tener errores de pegado, esto es, las fuerzas de pegado de cada lado del mosaico debe tener el mismo valor de sus mosaicos adyacentes, en cada una de sus orientaciones. La segunda restricción es que no tenga errores en la formación del patrón P.



$$Individuo = \begin{pmatrix} t_{1,1} & \cdots & t_{1,n} \\ \vdots & \ldots & \vdots \\ t_{m,1} & \cdots & t_{m,n} \end{pmatrix}$$

$$t_{1,2} = \{negro, 5{,}5, 3{,}3\} \qquad t_{i,j} = \{color, n, s, e, o\}$$

Figura 12: Representación del individuo en forma matricial, donde cada entrada en la matriz está ocupada por un mosaico para formar un patrón P dado.

Los individuos válidos son los que cumplen con las restricciones anteriores y son los que formarán parte de la población inicial. La generación de los individuos se denomina *Estricta*, ya que inicialmente cada individuo es una solución válida para el problema PATS.

3.3. Inicialización de la población

La inicialización de la población se realiza mediante la generación de un número previamente definido de individuos. Para generar cada uno de ellos, inicialmente se parte de un conjunto trivial de mosaicos, donde cada mosaico en el conjunto es diferente. Para generar el conjunto trivial de mosaicos, se produce de forma aleatoria un número que define la fuerza de pegado para cada una de las orientaciones del mosaico.

El conjunto trivial y el patrón (caso de prueba) se someten al auto-ensamblado para formar al individuo. Los mosaicos del conjunto trivial tienen un ligero cambio para cada individuo, que consiste en generar nuevos mosaicos para formar un nuevo conjunto trivial que mediante el auto-ensamblado constituya un nuevo individuo con el fin de dar aleatoriedad y variedad a la población inicial. Cada individuo generado se evalúa y, si cumple con las dos restricciones anteriormente descritas, forma parte de la población inicial (Figura 13(a)). En caso contrario se desecha. La generación de la población inicial finaliza, cuando se genera la cantidad de individuos predefinida (Algoritmo 2).

Algoritmo 2 InicializarPoblación

Entrada: conjunto trivial de mosaicos T y un patrón P.

Salida: conjunto de individuos válidos (poblacion).

1: **mientras** $poblacion < I_{max}$ **hacer**
2: individuo $\leftarrow auto - ensamblar(T, P)$;
3: valido$\leftarrow verificar($individuo$)$;
4: **si** valido **entonces**
5: poblacion$\leftarrow$ individuo;
6: **si no**
7: individuo$\leftarrow \varnothing$;
8: **fin si**
9: nuevosMosaicos $\leftarrow generaNuevoMosaico(T)$;
10: $T \leftarrow$ nuevosMosaicos;
11: **fin mientras**

Otras alternativas para generar la población inicial es alternando las dos restricciones previamente descritas. Por ejemplo, si un individuo auto-ensamblado no cumple con la restricción de formar el patrón P dado, pero sí cumple con no tener errores de pegado (Figura 13(b)), entonces este individuo se considera parte de la población, es decir, no se desecha. La otra forma es que los individuos auto-ensamblados que formen el patrón P dado, pero que tengan errores de pegado, se consideren parte de la población (Figura 13(c)).

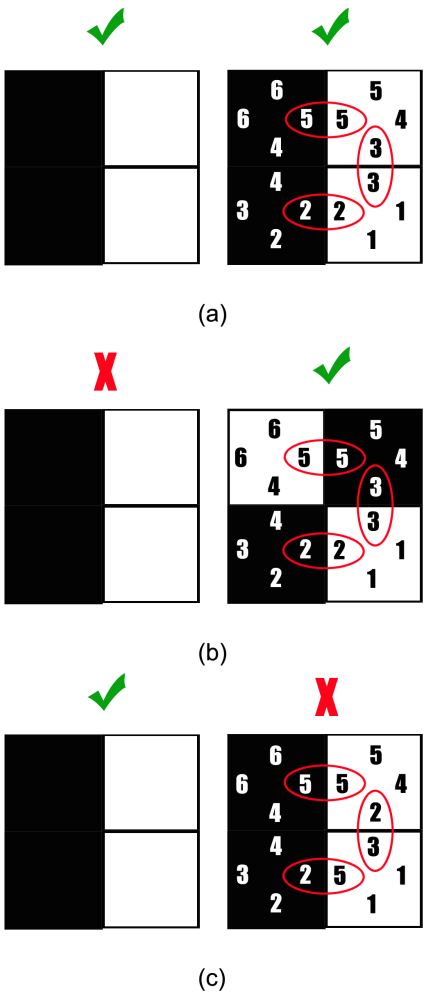


Figura 13: Formas de generar la población inicial: (a) modo estricto, (b) sin error de pegado, (c) que forme el patrón.

Cualquiera de las dos formas alternativas de generar la población inicial influye directamente con el mecanismo de evaluación de la función objetivo. Es decir, si los individuos válidos pueden formar un patrón P donde se permitan errores de pegado, entonces el mecanismo de evaluación de la función objetivo requiere de una restricción, que al momento de asignar la aptitud privilegie al individuo que no tenga errores de pegado. Sin embargo, si el individuo tiene errores de pegado no se desecha, sigue siendo un individuo válido pero con menos aptitud que uno sin errores de pegado.

La otra opción parte de que un individuo no tenga errores de pegado pero que no necesariamente forme el patrón. De igual forma, esto se ve reflejado en la evaluación de la función objetivo. El individuo que forme el patrón y no tenga errores de pegado

tendrá más aptitud que uno que no forme el patrón. No obstante, los individuos que no formen el patrón siguen siendo individuos válidos. Estas alternativas de inicialización de población las dejamos como trabajo a futuro. En esta tesis únicamente abarcamos la inicialización de la población de forma estricta.

3.4. Selección de padres

La selección de padres dentro de la población es por torneo binario. El torneo se lleva a cabo mediante la selección aleatoria de dos individuos de la población. Ambos individuos seleccionados se evalúan con la función objetivo y aquel que tenga menor cantidad de tipos de mosaicos recibe una aptitud más alta. El individuo con mayor aptitud se convierte en padre. Se realiza otro torneo para obtener al segundo padre y pasar a la siguiente etapa del algoritmo genético (Figura 8).

3.5. Operadores genéticos

Los individuos padres son sometidos a operadores genéticos. Los operadores genéticos se dividen en cruce y mutación. La función del operador de cruce es realizar combinaciones o cruces de genes dentro del cromosoma. En otras palabras, el operador de cruce intercambia filas o columnas de los padres para generar hijos con genes cruzados de los padres. La mutación se caracteriza por realizar un pequeño cambio en el cromosoma. Dichos cambios son transmitidos a su descendencia.

3.5.1. Operador de cruce

Los operadores de cruce presentados a continuación, fueron diseñados en (Cervantes-Salido *et al.*, 2013), (Gondro y Kinghorn, 2007) y (Wallet *et al.*, 1996), se adaptaron para este trabajo, de forma que cada entrada es un mosaico y no una letra.

3.5.1.1. Cruce horizontal

El operador de cruce horizontal toma las filas impares del Padre 1 y las pares del Padre 2 para generar el Hijo 1 y de forma inversa para generar al Hijo 2. El procedimiento describe a detalle en el Algoritmo 3, donde se requieren dos individuos como parámetros de entrada y tiene como salida dos individuos cruzados. El algoritmo inicia con el etiquetado de individuos (línea 1 y 2) donde se etiqueta a los individuos de entrada como Padre 1 y Padre 2. Posteriormente, se construyen dos individuos vacíos y se nombran Hijo 1 e Hijo 2 (líneas 3 y 4). De la línea 5 a la 12 se genera un ciclo anidado. En la línea 7 se inicializan los índices par e $impar$. En la línea 8 se realiza la asignación de las filas impares del Padre 1 al Hijo 1. Consecuentemente, en la línea 9 se completa la asignación del Hijo 1 con la asignación de las filas pares del Padre 2. En la línea 10 se asignan las filas pares del Padre 1 al Hijo 2. Posteriormente, en la línea 11 se asignan las filas impares del Padre 2 al Hijo 2 (Figura 14). De esta forma, se generan Hijo 1 e Hijo 2 que son los dos individuos cruzados horizontalmente que se obtienen de salida.

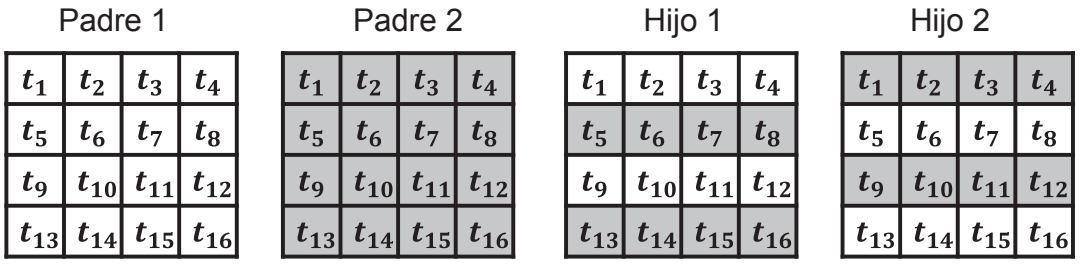


Figura 14: Operador de cruce horizontal.

3.5.1.2. Cruce vertical

El operador vertical genera un punto de cruce aleatorio y combina la primer parte del Padre 1 y la segunda parte del Padre 2 para crear al Hijo 1; de forma inversa genera el Hijo 2. En el Algoritmo 4 se describe al operador de cruce vertical detalladamente. Como parámetro de entrada toma dos individuos y como salida produce dos individuos cruzados verticalmente. El algoritmo inicia con la asignación de etiquetas para los individuos dados de entrada. En la línea 1 y 2 se etiquetan al Padre 1 y Padre 2, respectivamente.

Algoritmo 3 Cruce horizontal

Entrada: dos individuos (Padre 1 y Padre 2).

Salida: dos individuos hijos (Hijo 1 e Hijo 2).

1: Padre 1 $\leftarrow individuo$;
2: Padre 2 $\leftarrow individuo$;
3: Hijo 1 $\leftarrow \varnothing$;
4: Hijo 2 $\leftarrow \varnothing$;
5: **mientras** $i < longitud$ **hacer**
6: **mientras** $j < longitud$ **hacer**
7: par = 2j, $impar$ = 2j+1;
8: Hijo 1$[impar][j] \leftarrow$ Padre 1 $[impar][j]$;
9: Hijo 1$[par][j] \leftarrow$ Padre 2 $[par][j]$;
10: Hijo 2$[par][j] \leftarrow$ Padre 1 $[par][j]$;
11: Hijo 2$[impar][j] \leftarrow$ Padre 2 $[impar][j]$;
12: **fin mientras**
13: **fin mientras**

Posteriormente, en la línea 3 y 4 se crean dos individuos vacíos denominados Hijo 1 e Hijo 2. En la línea 5 se genera un punto de cruce de forma aleatoria, este punto de cruce sirve como pivote para dividir a los individuos Padre 1 y Padre 2 (Figura 15). Durante el ciclo 6-11 se realiza la asignación de la primera parte delimitada por el pivote: en la línea 8 se asigna la primera parte del Padre 1 al Hijo 1 y en la línea 9 se asigna la primera parte del Padre 2 al Hijo 2. De las líneas 12 a la 17 se asigna la segunda parte que divide el pivote: En la línea 14 se asigna la segunda parte del Padre 2 al Hijo 1 y en la línea 15 la segunda parte del Padre 1 se asigna al Hijo 2. La salida del algoritmo son Hijo 1 e Hijo 2, dos individuos cruzados verticalmente.

En base a observación del comportamiento de los operadores de cruce por fila y vertical se llegó a la conclusión que es posible unirlos para formar un operador genérico que el cruce este en función de los índices matriciales.

Algoritmo 4 Cruce vertical

Entrada: dos individuos (Padre 1 y Padre 2).

Salida: dos individuos hijos (Hijo 1 e Hijo 2).

1: Padre 1 $\leftarrow individuo$;
2: Padre 2 $\leftarrow individuo$;
3: Hijo 1 $\leftarrow \varnothing$;
4: Hijo 2 $\leftarrow \varnothing$;
5: $pivote = Random(longitud)$
6: **para** $i = 0$ **to** $i < longitud$ **hacer**
7: **para** $j = 0$ **to** $j < pivote$ **hacer**
8: Hijo 1$[i][j] \leftarrow$ Padre 1$[i][j]$;
9: Hijo 2$[i][j] \leftarrow$ Padre 2$[i][j]$;
10: **fin para**
11: **fin para**
12: **para** $i = 0$ **to** $i < longitud$ **hacer**
13: **para** $j = pivote$ **to** $j < longitud$ **hacer**
14: Hijo 1$[i][j] \leftarrow$ Padre 2$[i][j]$;
15: Hijo 2$[i][j] \leftarrow$ Padre 1$[i][j]$;
16: **fin para**
17: **fin para**

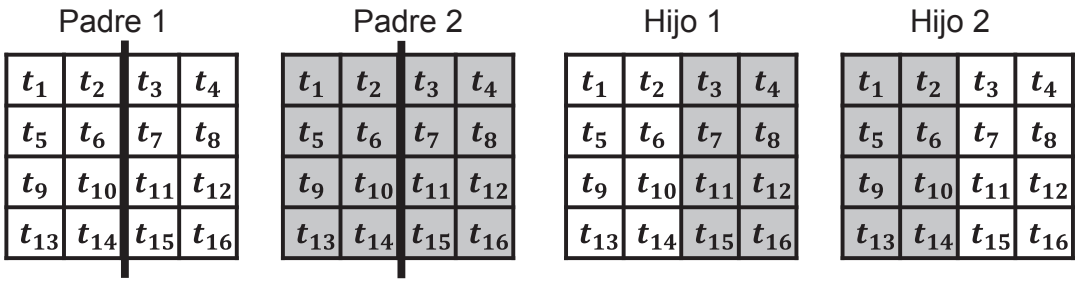


Figura 15: Operador de cruce vertical. La línea negra corresponde al punto de cruce, el cual es generado aleatoriamente.

3.5.1.3. Cruce por marco

El operador de cruce por marco genera un marco en el individuo, es decir, selecciona los mosaicos del contorno del individuo y realiza una combinación de los padres para generar a los hijos. El operador se describe en el Algoritmo 5, donde la entrada son dos individuos y la salida dos individuos cruzados por marco. El algoritmo inicia con la asignación de etiquetas: las líneas 1 y 2 etiquetan como Padre 1 y Padre 2 a los dos individuos dados como entrada. En la línea 3 y 4 se crean dos individuos vacíos etiquetados como Hijo 1 e Hijo 2. En el ciclo 5 -14 se generan los marcos, los cuales dependen de la longitud del individuo. En la línea 6 se toma la última columna del Padre 1 y se asigna al Hijo 1. Luego, se toma la primer columna del Padre 1 y se asigna al Hijo 1 (línea 7). Después, en la línea 8 se toma la primer fila del Padre 1 y se asigna al Hijo 1. Para finalizar el marco, se toma la última fila del Padre 1 y se asigna al Hijo 1 (línea 9). Las siguientes líneas (10 - 13), realizan el mismo procedimiento descrito anteriormente, pero ahora del Padre 2 al Hijo 2. En las líneas 15 a la 20 se realiza la asignación del centro de los marcos (Figura 16): La línea 17 asigna el centro del marco del Padre 2 al Hijo 1 y en la línea 18 se asigna el centro del marco del Padre 1 al Hijo 2. La salida del algoritmo son Hijo 1 e Hijo 2 que son individuos cruzados por marco.

3.5.2. Operador de mutación

El operador de mutación se aplica a cada hijo de manera individual, y consiste en la alteración aleatoria de los genes del cromosoma, normalmente con probabilidad pequeña. Si bien puede en principio pensarse que el operador de cruce es más importante que el operador de mutación, al proporcionar una exploración rápida del espacio de búsqueda,

Algoritmo 5 Cruce de marco

Entrada: dos individuos (Padre 1 y Padre 2).

Salida: dos individuos hijos (Hijo 1 e Hijo 2).

1: Padre 1 $\leftarrow individuo$;
2: Padre 2 $\leftarrow individuo$;
3: Hijo 1 $\leftarrow \varnothing$;
4: Hijo 2 $\leftarrow \varnothing$;
5: **para** $i = 0$ **to** $i < longitud$ **hacer**
6: Hijo 1$[0][i] \leftarrow$ Padre 1$[0][i]$;
7: Hijo 1$[i][0] \leftarrow$ Padre 1$[i][0]$;
8: Hijo 1$[longitud][i] \leftarrow$ Padre 1$[longitud][i]$;
9: Hijo 1$[i][longitud] \leftarrow$ Padre 1$[i][longitud]$;
10: Hijo 2$[0][i] \leftarrow$ Padre 2$[0][i]$;
11: Hijo 2$[i][0] \leftarrow$ Padre 2$[i][0]$;
12: Hijo 2$[longitud][i] \leftarrow$ Padre 2$[longitud][i]$;
13: Hijo 2$[i][longitud] \leftarrow$ Padre 2$[i][longitud]$;
14: **fin para**
15: **para** $i = 0$ **to** $i < longitud - 1$ **hacer**
16: **para** $j = 0$ **to** $j < longitud - 1$ **hacer**
17: Hijo 1$[i][j] \leftarrow$ Padre 2$[i][j]$;
18: Hijo 2$[i][j] \leftarrow$ Padre 1$[i][j]$;
19: **fin para**
20: **fin para**

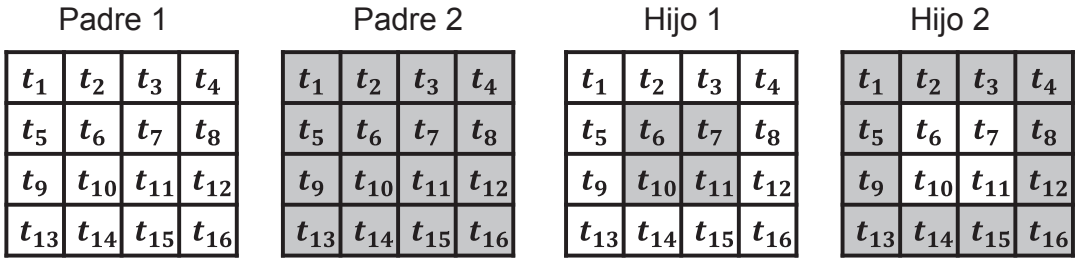


Figura 16: Operador de cruce de marco.

este último asegura que ningún punto del espacio de búsqueda tenga probabilidad cero de ser examinado, y es de gran importancia para asegurar la convergencia del algoritmo genético (Eiben y Smith, 2003). En esta tesis se consideran tres operadores de mutación diseñados en este trabajo para el problema PATS.

3.5.2.1. Mutación por color

El operador de mutación por color selecciona una fila del individuo de forma aleatoria; selecciona un color (blanco o negro) aleatoriamente y muta todos los mosaicos del color seleccionado que tenga la fila (Figura 17). En el Algoritmo 6 se describe al operador de mutación por color. Este algoritmo toma como parámetro de entrada a un individuo y como salida se obtiene un individuo mutado por color. El proceso inicia en la línea 1, generando un número de fila del individuo de manera aleatoria (el número depende del tamaño del individuo). En la línea 2 se genera un bit aleatorio (0 ó 1) para determinar el color a mutar (blanco o negro). Luego, se crea una copia del individuo a mutar y se asigna a la estructura “individuoMutado” que servirá como individuo de entrada (línea 3). Posteriormente, se detectan los mosaicos de la fila seleccionada aleatoriamente, con el color seleccionado, también aleatoriamente (líneas 4 - 8). Estos mosaicos se almacenan en un arreglo llamado *mosaicos*.

En la línea 9 se manda llamar una función denominada *generaNuevoMosaico*, la cual toma como parámetro de entrada el arreglo *mosaicos*. Las operaciones que realiza esta función se ilustran en la Figura 18. En el inciso (a) se muestra el conjunto de mosaicos que anteriormente fue seleccionado de la fila del individuo. En el inciso (b) se ilustra la

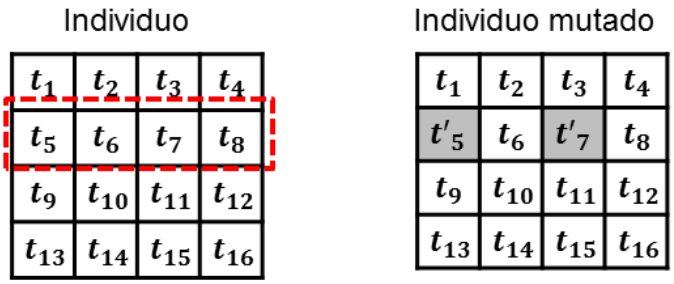


Figura 17: Operador de mutación por color. El recuadro de líneas punteadas representa la fila seleccionada de forma aleatoria. Los mosaicos sombreados representan a los mosaicos que son afectados por la mutación.

separación por fuerzas de pegado, que se realiza partiendo de que en los mosaicos únicamente se pueden unir las orientaciones $norte \leftrightarrow sur$ y $este \leftrightarrow oeste$, establecido en el modelo aTAM donde los mosaicos no pueden rotarse (Winfree *et al.*, 1998), por lo que las fuerzas de pegado de los mosaicos se agrupan en parejas ($(norte, oeste)$ y $(sur, este)$). En el inciso (c) se muestra que partiendo de un mosaico se pueden crear dos, asignándoles las fuerzas de pegado de $(norte, oeste)$ a uno y del $(sur, este)$ al otro. Finalmente, en el inciso (d) se ilustra el conjunto de mosaicos generados partiendo de las parejas de fuerzas de pegado, en donde la fuerza de pegado del $norte$ tiene el mismo número que el sur; de igual forma pasa para las fuerzas de pegado del $este$ y del $oeste$ dentro del mismo mosaico.

De este conjunto de nuevos mosaicos se selecciona uno de ellos de forma aleatoria. El mosaico seleccionado es el que regresa la función *generaNuevoMosaico*, asignándose a la variable *mosaicoNuevo*. En las líneas 10 a la 14 se reemplazan los mosaicos seleccionados inicialmente en el “individuoMutado”, por el mosaico almacenado en *mosaicoNuevo*. De esta forma el algoritmo produce como salida un individuo mutado por color.

3.5.2.2. Mutación por fila

El operador por fila selecciona aleatoriamente una fila del individuo y muta todos los mosaicos de la fila (blancos y negros), mediante la generación de nuevos mosaicos. En el Algoritmo 7 se describe a detalle. Este procedimiento toma como entrada un indivi-

Algoritmo 6 Mutación por color

Entrada: individuo.

Salida: individuoMutado.

1: $fila \leftarrow Random(longitud)$;
2: $color \leftarrow Random(2)$;
3: individuoMutado $\leftarrow$ individuo;
4: **para** $i = 0$ **to** $i < longitud$ **hacer**
5: **si** individuo$[i][fila] = color$ **entonces**
6: $mosaicos[i] \leftarrow$ individuo$[i][fila]$;
7: **fin si**
8: **fin para**
9: $mosaicoNuevo \leftarrow generaNuevoMosaico(mosaicos[])$;
10: **para** $i = 0$ **to** $i < longitud$ **hacer**
11: **si** individuo$[i][fila] = color$ **entonces**
12: individuoMutado$[i][fila] \leftarrow mosaicoNuevo$;
13: **fin si**
14: **fin para**

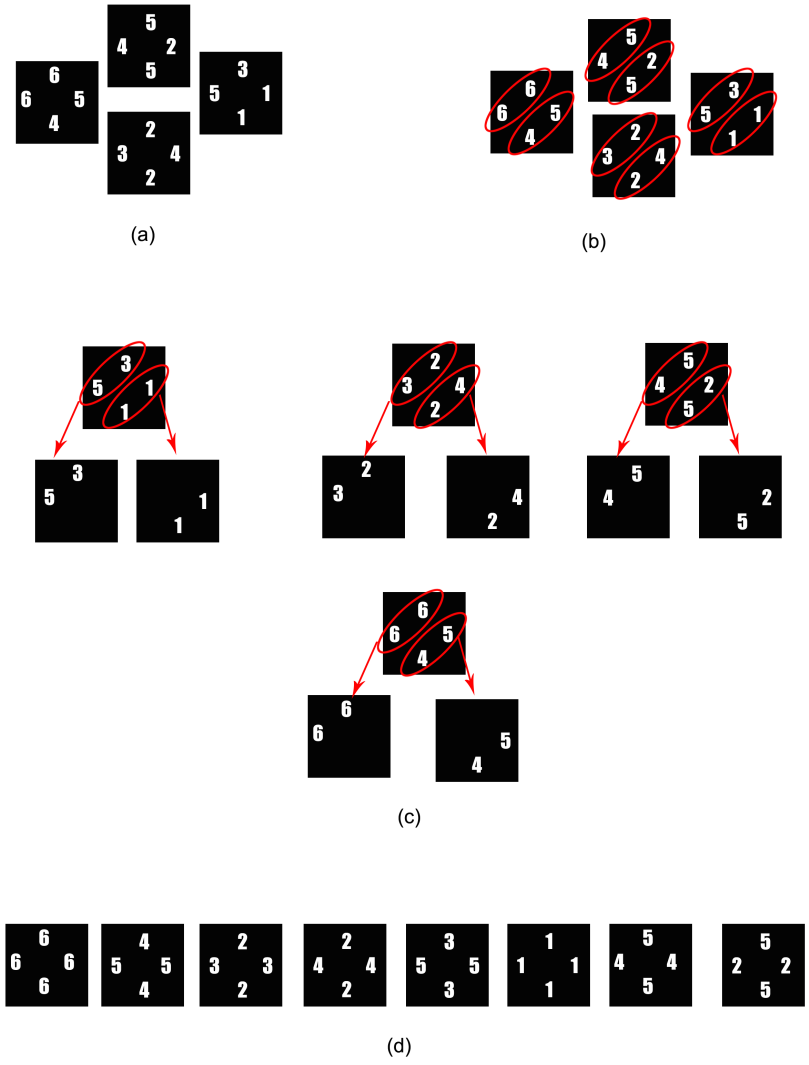


Figura 18: Generación de nuevos mosaicos: (a) conjunto de mosaicos seleccionados, (b) agrupación de fuerzas de pegado por pares $(norte, oeste)$ y $(sur, este)$, (c) asignación de fuerzas de pegado por pares a los mosaicos generados, (d) conjunto de nuevos mosaicos.

duo y provee como salida el individuo mutado por fila. El algoritmo inicia generando un número correspondiente a una fila del individuo de forma aleatoria (línea 1). Después se inicializan las variables que se encargan de almacenar el color del mosaico (líneas 2 - 3). En las líneas 4 y 5 se inicializan dos arreglos para almacenar los mosaicos de la fila seleccionada con anterioridad, y con base al color del mosaico se almacena en su respectivo arreglo. En las líneas 6 y 7 se inicializan *mosaicoNuevoB* y *mosaicoNuevoN* que son dos mosaicos que se generarán más adelante. En la línea 9 se inicia con un ciclo que recorre al individuo. En el ciclo 10-14 se seleccionan los mosaicos de la fila seleccionada en la línea 1 (estos mosaicos se almacenan en los arreglos dependiendo del color). Pos-

teriormente, se manda llamar a la función *generaNuevoMosaico* para generar un nuevo mosaico blanco y un nuevo mosaico negro (líneas 16 - 17). La función *generaNuevoMosaico* se describe en la Figura 18, esta función regresa *nuevoMosaicoB* en la línea 16 y *nuevoMosaicoN* en la línea 17. En las líneas 18-24 se reemplazan los mosaicos de la fila seleccionada anteriormente en la línea 1, por los nuevos mosaicos. El reemplazo mantiene el patrón formado inicialmente en el individuo dado de entrada. De esta forma el algoritmo regresa un "individuoMutado" por fila (Figura 19).

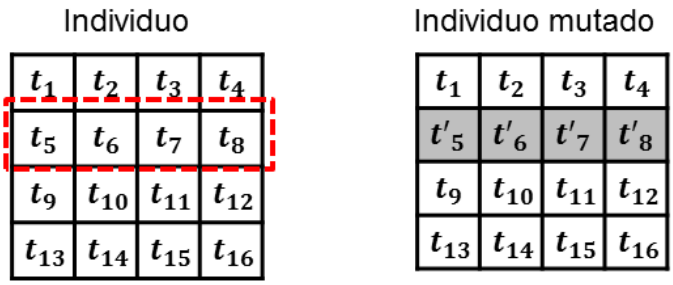


Figura 19: Operador de mutación por fila. El recuadro de líneas punteadas representa la fila seleccionada de forma aleatoria. Los mosaicos sombreados representan a los mosaicos que son afectados por la mutación.

3.5.2.3. Mutación por máscara

Este operador de mutación genera una máscara de forma aleatoria, y los mosaicos afectados en el individuo por la aplicación de la máscara son mutados (Figura 20). El Algoritmo 8 describe detalladamente al operador, el cual toma como parámetro de entrada a un individuo y lo regresa de forma mutada por máscara. El algoritmo comienza con la inicialización de las variables requeridas para generar la máscara y arreglos de almacenamiento (líneas 1-8). En las líneas 8 a la 17 se genera la máscara aleatoriamente y sus mosaicos se almacenan en los arreglos dependiendo de su color. En las líneas 18 y 19 se manda llamar a la función *generaNuevoMosaico* (Figura 18), que regresa un nuevo mosaico. En la línea 18 se almacena el nuevo mosaico en *nuevoMosaicoB* y en la línea 19 se almacena en *nuevoMosaicoN*. Por último, se aplica la máscara y se reemplazan los mosaicos del individuo por los de la máscara, conservando el patrón formado en el individuo de entrada (líneas 20 - 30). De esta forma el algoritmo regresa un individuoMutado por máscara.

Algoritmo 7 Mutación por fila

Entrada: individuo.

Salida: individuoMutado.

1: $fila \leftarrow Random(longitud)$;
2: $blanco \leftarrow 0$;
3: $negro \leftarrow 1$
4: $mosaicosBlancos \leftarrow \varnothing$;
5: $mosaicosNegros \leftarrow \varnothing$;
6: $mosaicoNuevoB \leftarrow \varnothing$;
7: $mosaicoNuevoN \leftarrow \varnothing$;
8: individuoMutado ← individuo;
9: **para** $i = 0$ **to** $i < longitud$ **hacer**
10: **si** individuo$[i][fila] = blanco$ **entonces**
11: $mosaicosBlancos[i] \leftarrow$ individuo$[i][fila]$;
12: **si no**
13: $mosaicosNegros[i] \leftarrow$ individuo$[i][fila]$;
14: **fin si**
15: **fin para**
16: $mosaicoNuevoB = generaNuevoMosaico(mosaicosBlancos[])$
17: $mosaicoNuevoN = generaNuevoMosaico(mosaicosNegros[])$;
18: **para** $i = 0$ **to** $i < longitud$ **hacer**
19: **si** individuo$[i][fila] = blanco$ **entonces**
20: individuoMutado$[i][fila] \leftarrow mosaicoNuevoB$;
21: **si no**
22: individuoMutado$[i][fila] \leftarrow mosaicoNuevoN$;
23: **fin si**
24: **fin para**

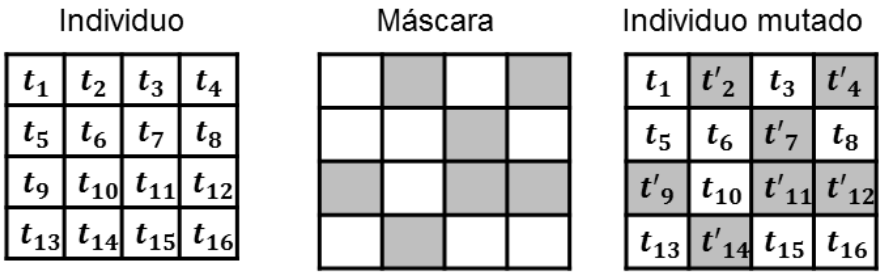


Figura 20: Operador de mutación por máscara. Aquellos mosaicos que coincidan con los de la máscara son mutados. Los mosaicos sombreados representan a los mosaicos que son mutados utilizando la función *generaNuevoMosaico*.

En base a observación del funcionamiento de los tres operadores de mutación se llegó a la conclusión que es posible unirlos para formar un operador genérico que este en función de la máscara producida.

Es importante notar que los individuos resultantes de los operadores genéticos, no necesariamente son factibles, es decir, se requiere verificarlos para determinar si cumplen las restricciones establecidas inicialmente para formar parte de la población.

3.6. Selección de sobrevivientes

En nuestro algoritmo, después de aplicar los operadores de variación se generan hijos, los cuales se someten a una verificación que consiste en corroborar que no contengan errores de pegado y que el individuo construya el patrón P dado. En caso de no cumplir con la verificación se elige otro individuo de la población hasta generar hijos válidos. Los hijos válidos son evaluados con la función objetivo para posteriormente seleccionar a los que son capaces de sobrevivir. La selección de sobrevivientes se realiza por medio del valor de aptitud del individuo: Aquellos con mejor aptitud (con menor cantidad de tipo de mosaicos) son seleccionados para formar parte de la siguiente generación.

3.7. Mecanismo de evaluación de la función objetivo

Los individuos se evalúan con la función objetivo mediante un mecanismo que permite definir la aptitud de cada individuo evaluado. El mecanismo de evaluación de la función

Algoritmo 8 Mutación por máscara

Entrada: individuo.

Salida: individuoMutado.

1: $mascara \leftarrow \varnothing$; $bandera \leftarrow 1$;
2: $blanco \leftarrow 0$; $negro \leftarrow 1$
3: $x \leftarrow \varnothing$; $y \leftarrow \varnothing$;
4: $mosaicosBlancos \leftarrow \varnothing$; $mosaicosNegros \leftarrow \varnothing$;
5: $mosaicoNuevoB \leftarrow \varnothing$; $mosaicoNuevoN \leftarrow \varnothing$;
6: $cantidad \leftarrow \lceil longitud \setminus 2 \rceil$;
7: individuoMutado ← individuo;
8: **para** $i = 0$ **to** $i < cantidad$ **hacer**
9: $x \leftarrow Random(longitud)$;
10: $y \leftarrow Random(longitud)$;
11: $mascara[x][y] \leftarrow bandera$;
12: **si** individuo$[x][y] = blanco$ **entonces**
13: $mosaicosBlancos[i] \leftarrow$ individuo$[x][y]$;
14: **si no**
15: $mosaicosNegros[i] \leftarrow$ individuo$[x][y]$;
16: **fin si**
17: **fin para**
18: $mosaicoNuevoB \leftarrow generaNuevoMosaico(mosaicosBlancos[])$;
19: $mosaicoNuevoN \leftarrow generaNuevoMosaico(mosaicosNegros[])$;
20: **para** $i = 0$ **to** $i < longitud$ **hacer**
21: **para** $j = 0$ **to** $j < longitud$ **hacer**
22: **si** $mascara[i][j] = bandera$ **entonces**
23: **si** individuo$[i][j] = blanco$ **entonces**
24: individuoMutado$[i][j] \leftarrow mosaicoNuevoB$;
25: **si no**
26: individuoMutado$[i][j] \leftarrow mosaicoNuevoN$;
27: **fin si**
28: **fin si**
29: **fin para**
30: **fin para**

objetivo toma como entrada un individuo, el cual tiene una estructura matricial. La matriz contiene $N \times N$ mosaicos, cada mosaico tiene un color y fuerzas de pegado en cada uno de sus cuatro lados. El mecanismo recorre cada uno de los mosaicos en la matriz y los etiqueta con un tipo con base en la combinación de su color y fuerzas de pegado. Los almacena en una lista de tipos, llevando la contabilidad de los tipos existentes en el individuo. Al finalizar el recorrido por toda la matriz, el mecanismo devuelve el número de tipos de mosaicos del mismo (la aptitud del individuo).

Partiendo de que un individuo dado como parámetro de entrada es válido, es decir, cumple con formar el patrón (caso de prueba) sin contener errores de pegado, se puede asegurar que el individuo después de ser evaluado sigue siendo válido.

3.8. Conjunto de parámetros

La configuración de parámetros de un algoritmo genético es difícil de concertar, ya que el número de dichas configuraciones es variado y cada una de ellas tiene un intervalo significativo de ajuste.

El conjunto de parámetros que maneja el AG-PATS se enlista a continuación:

- Tamaño de la población
- Número máximo de generaciones
- Probabilidad de cruce
- Probabilidad de mutación
- Tipo de cruce
- Tipo de mutación

La finalidad del conjunto de parámetros es encontrar una configuración adecuada, que sea capaz de obtener resultados novedosos para cada uno de los diferentes casos de prueba. Teniendo en cuenta que son 16 los casos de prueba existentes en la literatura, encontrar una configuración que sea conveniente para todos ellos, tiene un mayor grado de complejidad.

3.9. Algoritmo genético para el problema PATS (AG-PATS)

El Algoritmo 9 corresponde al esquema del algoritmo genético simple. Se consideró este esquema dado que estamos tratando un problema de optimización mono-objetivo. Los algoritmos genéticos tienen diferentes enfoques y se especializan al tipo de problema con el que se está tratando. Son capaces de solucionar problemas mono y multi-objetivo, realizando variaciones al algoritmo genético pero sin perder el esquema general. Para solucionar el problema PATS utilizamos el esquema simple. Sin embargo, hacemos uso de métodos especializados que proveen las condiciones necesarias para solucionar el problema. A continuación se presenta el pseudocódigo del algoritmo.

Algoritmo 9 Algoritmo genético para el problema PATS (AG-PATS)

Entrada: conjunto trivial de mosaicos T y un patrón P.

Salida: un individuo que es un sistema de mosaicos **T** $= \langle T, S, g, \tau \rangle$ donde T es el conjunto mínimo de tipos de mosaicos y **T** auto-ensamble el patrón P.

1: poblacion $\leftarrow inicializarPoblacion()$;
2: poblacion$\leftarrow mecanismoDeEvaluacion$(poblacion);
3: **mientras** $generaciones < I_{max}$ **hacer**
4: padres$\leftarrow seleccion$(poblacion);
5: hijos$\leftarrow cruzar$(padres);
6: poblacion$\leftarrow mutar$(hijos);
7: poblacion$\leftarrow mecanismoDeEvaluacion$(poblacion);
8: poblacion$\leftarrow seleccion$(poblacion);
9: **fin mientras**
10: individuo$\leftarrow seleccion$(poblacion);

Iniciamos con la descripción del Algoritmo 9, donde los parámetros de entrada son: un patrón P, que es el que define el caso de prueba a tratar, un conjunto trivial de mosaicos T, tamaño de la población, número de generaciones, probabilidad de cruce, probabilidad de mutación, tipo de cruce y tipo de mutación. El algoritmo proporciona como salida un sistema de mosaicos **T** $= \langle T, S, g, \tau \rangle$, donde T es el conjunto de tipos de mosaicos optimizado.

En la primer línea, se genera la población inicial de individuos. Para esto, se requiere un método especializado que se describe a detalle en el Algoritmo 2. Posteriormente, en la línea 2, se realiza la evaluación de los individuos que forman parte de la población inicial. El mecanismo de evaluación de la función objetivo es el encargado de evaluar y asignar la aptitud a cada individuo. La función objetivo para el problema PATS es minimizar el tipo de mosaicos en el conjunto. Con base en esto, el mecanismo de evaluación asigna la aptitud a cada individuo. Dependiendo de la cantidad de tipos de mosaicos es la aptitud asignada. Por ejemplo, si el tamaño de un individuo es de 8×8, su conjunto trivial es de 64 mosaicos y por tanto su aptitud de 64. Suponga que el individuo ya ha sido evolucionado un par de generaciones. Entonces, si el tipo de mosaicos es menor que 64, la aptitud será también menor. Como es un problema de minimización, entre menos sea la aptitud, más apto es el individuo para sobrevivir.

En la línea 4, se realiza la selección de padres (Sección 3.4). A este par de individuos seleccionados se le aplica los operadores genéticos. El primer operador es el cruce (línea 5). La forma de cruce está definida mediante el operador de cruce dado como parámetro de entrada. Después, los hijos generados por el cruce son mutados, donde se convierten en individuos mutados mejor conocidos como descendencia (línea 6). En la línea 7, se evalúa la descendencia con el mecanismo de la función objetivo, la cual les asigna su aptitud como individuos de la población. Por último, se realiza la selección de sobrevivientes (Sección 3.6); los individuos más aptos pasaran a formar parte de la población de la siguiente generación (línea 8). El proceso evolutivo finaliza cuando el número de generaciones previamente definido ha sido completado.

Es importante notar que, en AG-PATS, los operadores de mutación tienen gran influencia. De acuerdo a observaciones experimentales, los operadores de mutación anteriormente descritos realizan cambios significativos en la evolución de la población, impactando principalmente en la convergencia de la misma.

En este capítulo se presentó el diseño del AG-PATS, en el siguiente capítulo (Capítulo 4) se muestra la experimentación del mismo y los resultados obtenidos.

Capítulo 4. Experimentos y resultados

En este capítulo se presentan las especificaciones computacionales que se requirieron en la experimentación del algoritmo genético, así como los casos de prueba, resultados preliminares y las diferentes configuraciones de parámetros. Finalmente, se presentan los resultados obtenidos de la experimentación exhaustiva del algoritmo genético para el problema PATS.

4.1. Especificaciones computacionales

El equipo de cómputo utilizado para el desarrollo del algoritmo genético para el problema PATS es el siguiente. El lenguaje de programación utilizado para codificar el algoritmo genético es Java, utilizando como entorno de desarrollo NetBeans IDE. Se eligió este lenguaje de programación por sus múltiples ventajas, de entre ellas la escalabilidad que proporciona. Las ejecuciones del algoritmo genético se efectuaron en una iMac con un procesador Intel Core i5 2.7 Ghz, memoria RAM DDR3 de 4GB y disco HDD de 1TB. También se hizo uso de un servidor Mac Pro con procesador Intel Xeon Dual Core 2.6 Ghz, memoria RAM de 64GB y disco HDD de 1TB. Para realizar lectura de tiempo de procesamiento se utilizó la computadora iMac.

4.2. Casos de prueba

Los casos de prueba utilizados para evaluar el desempeño del algoritmo genético propuesto fueron tomadas de la literatura. Se agrupan en patrones de dos dimensiones, tales como el contador binario (*Binary counter*), el tablero de ajedrez (*Chess board*), la esquina de árbol (*Corner tree*), el patrón aleatorio (*Random*) y el triángulo de Sierpinski. Cada uno de ellos contienen instancias de diferentes tamaños, con un total de 16 casos de prueba diferentes. A continuación se describen a detalle.

4.2.1. Patrón contador binario

El patrón de contador binario mejor conocido como *Binary counter* fue introducido por Ma y Lombardi (2008), el cual simula un contador binario. El objetivo del contador es simplemente contar 1, 2, 3, 4, 5, ..., pero en base 2, es decir, 1, 10, 11, 100, 101, ... La forma de los mosaicos representa la información usada en el cómputo: los lados de arriba y de abajo codifican el valor del bit en el contador, mientras que los lados de la derecha y de la izquierda se usan para indicar si el bit debe o no cambiar su valor. El mosaico de color negro indica que el bit es 1 y el mosaico de color blanco que el bit es 0 (Figura 21). El tamaño de la instancia del patrón determina hasta donde va a llegar el contador binario. El patrón *Binary counter* cuenta con diferentes instancias que varían sus tamaños de 6×6, 15×15 y 32×32.

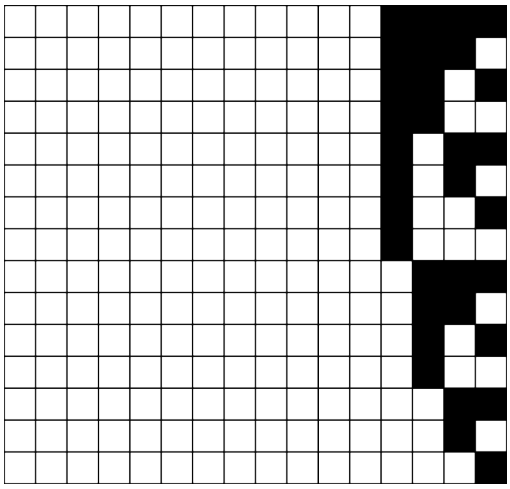


Figura 21: Patrón *Binary counter* de 15×15.

4.2.2. Patrón esquina de árbol

El patrón esquina de árbol (*Corner tree*) fue introducido por Lempiäinen *et al.* (2011), consiste de un patrón de 23×23 mosaicos (Figura 22). La figura que forma este patrón es un árbol en una esquina, donde la raíz se encuentra en la esquina izquierda inferior y las ramas se extienden diagonalmente. De todos los casos de prueba, este patrón es el más nuevo y por ello el menos experimentado.

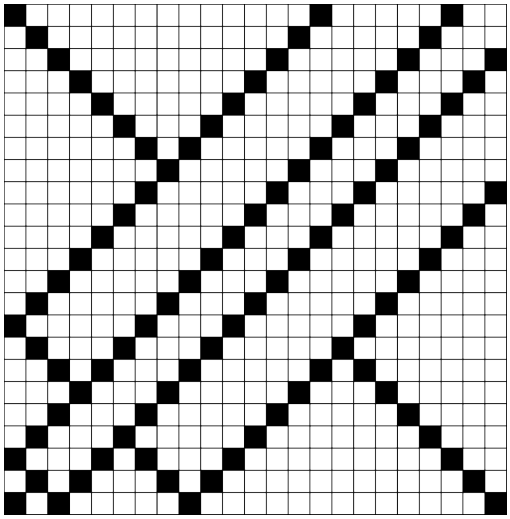


Figura 22: Patrón *Corner tree* de 23×23.

4.2.3. Patrón tablero de ajedrez

El patrón tablero de ajedrez mejor conocido como *Chess board* es uno de los más conocidos, al igual que *Binary counter*, fue introducido por (Ma y Lombardi, 2008). La estructura de este patrón se describe como una superficie de $n \times n$ que está compuesta de mosaicos blancos y negros que se intercalan entre ellos (Figura 23). El grupo de *Chess board* contiene dos patrones de diferentes tamaños, uno de 6×6 y otro de 15×15 mosaicos.

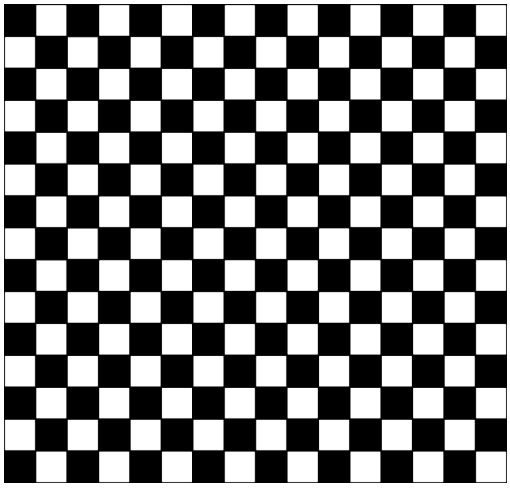


Figura 23: Patrón *Chess board* de 15×15.

4.2.4. Patrón aleatorio

El patrón aleatorio (*Random*) fue introducido por Göös y Orponen (2010), la forma para generar el patrón es la siguiente: se genera una matriz de ceros, posteriormente se generan coordenadas (x, y) de forma aleatoria, que reemplazan con 1 en la posición de la matriz que inicialmente estaba en 0. El tamaño de la matriz depende del tamaño de la instancia del patrón. De esta forma se genera el patrón *Random* donde los ceros de la matriz son representados por mosaicos de color blanco y los unos son representados por mosaicos de color negro. En la Figura 24 se describe el patrón *Random* utilizado en el algoritmo genético para el problema PATS. El patrón *Random* tiene diferentes instancias de 12×12, 16×16, 20×20 y 32×32.

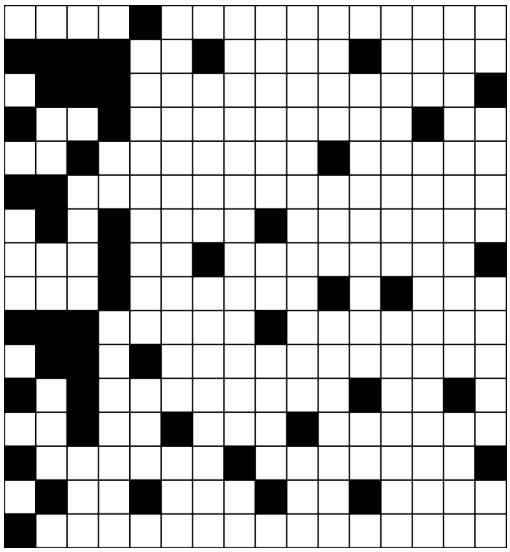


Figura 24: Patrón *Random* de 16×16.

4.2.5. Triángulo de Sierpinski

El triángulo de Sierpinski es de los primeros patrones utilizados en la literatura de auto-ensamblado de ADN. Entre los trabajos pioneros se encuentra (Rothemund *et al.*, 2004), utilizan el triángulo de Sierpinski como patrón a formar mediante este modelo. La técnica que usan es un autómata celular para actualizar la regla que calcula la función XOR binaria y así se va auto-ensamblando el patrón fractal. Los resultados de este trabajo es un ensamblaje que forma el patrón del triángulo de Sierpinski en laboratorio experimental. Otro de los trabajos que usa este patrón es (Ma y Lombardi, 2008), donde

es el patrón principal en los casos de prueba del problema PATS. Este patrón tiene distintas instancias (6×6, 8×8, 12×12, 17×17 y 32×32) introducidas por Ma y Lombardi (2008), posteriormente Lempiäinen *et al.* (2011) propone el de 64×64. Esto lo hace el patrón más numeroso y con la instancia más grande de todos los patrones utilizados en el estado del arte. En la Figura 25 podemos observar el patrón con la instancia de 17×17.

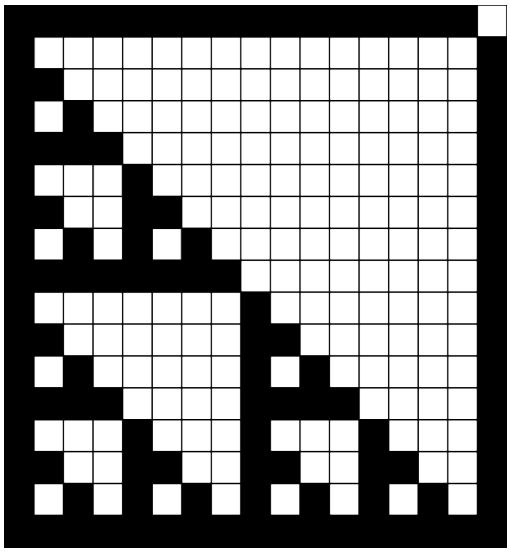


Figura 25: Patrón triángulo de Sierpinski de 17×17.

4.3. Resultados preliminares

Las primeras pruebas realizadas al AG-PATS utilizaron un conjunto de parámetros, usualmente empleados en algoritmos genéticos. Se consideró una población inicial de 50 individuos y 500 generaciones, usando un operador de cruce con probabilidad de 0.6 y otro de mutación con probabilidad de 0.1. El tipo de operador de cruce es el cruce por fila y en la mutación por color.

Con esta configuración de parámetros, se ejecutó el algoritmo genético 10 veces para un caso de prueba (Sierpinski 8×8). En la gráfica se muestra el promedio de las ejecuciones del algoritmo genético (Figura 26). El objetivo de estas pruebas fue mostrar que el algoritmo diseñado funciona para solucionar el problema PATS.

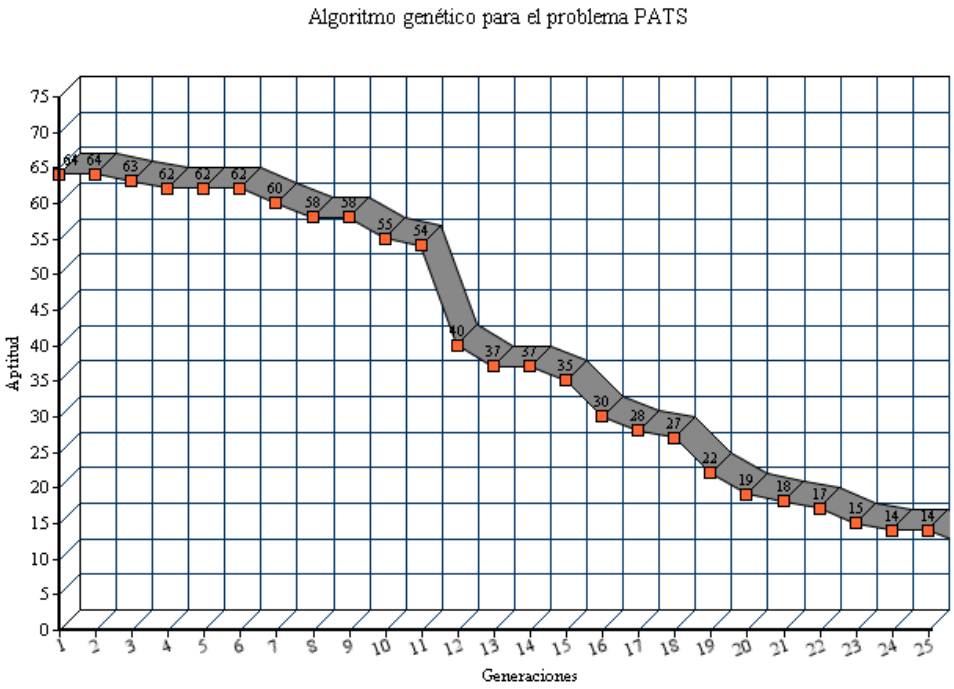


Figura 26: Resultados preliminares para el patrón de Sierpinski con instancia de 8×8.

Tabla 1: Comparación de resultados preliminares del AG-PATS con las propuestas del estado del arte: [1] Ma y Lombardi (2008), [2] Göös y Orponen (2010) y [3] Lempiäinen *et al.* (2011). En negritas se identifica la mejor solución.

Patrón	Tamaño	Conjunto trivial	Voraz [1]	B&B [2]	PS-BB [3]	AG-PATS
Binary counter	6×6	36	19	$n \setminus a$	$n \setminus a$	**4**
	15×15	225	121	$n \setminus a$	$n \setminus a$	**17**
	32×32	1024	$n \setminus a$	62	20	**20**
Corner tree	23×23	529	$n \setminus a$	$n \setminus a$	**25**	35
Chess board	6×6	36	19	$n \setminus a$	$n \setminus a$	**9**
	15×15	225	119	$n \setminus a$	$n \setminus a$	**32**
Random	12×12	144	$n \setminus a$	58	$n \setminus a$	**19**
	16×16	256	$n \setminus a$	100	$n \setminus a$	**23**
	20×20	400	$n \setminus a$	160	$n \setminus a$	**33**
	32×32	1024	$n \setminus a$	420	$n \setminus a$	**69**
Sierpinski	6×6	36	21	$n \setminus a$	$n \setminus a$	**7**
	8×8	64	37	$n \setminus a$	$n \setminus a$	**13**
	12×12	144	80	20	$n \setminus a$	**20**
	17×17	289	172	23	$n \setminus a$	**21**
	32×32	1024	595	60	42	**40**
	64×64	4096	$n \setminus a$	$n \setminus a$	95	**82**

Para esto se realizaron también experimentaciones con diferente configuración de parámetros, guiadas en base a prueba y error. La configuración de parámetros con mejores resultados en esta etapa fue la siguiente: una población de 50 individuos con 150,000 generaciones, una probabilidad de cruce de 0.6 y utilizando el operador de cruce vertical; la probabilidad de mutación fue de 0.5 con el operador de mutación por fila. En la Tabla 1 se muestran los resultados preliminares del algoritmo genético para el problema PATS, los cuales se comparan con los resultados de otras propuestas del estado del arte. Se puede observar la reducción del conjunto de mosaicos de cada propuesta, la cual se realiza partiendo de un conjunto trivial de mosaicos. La máxima reducción se muestra con negritas. Con base en esto, se hace un análisis comparativo entre los resultados expuestos. Podemos apreciar que los resultados obtenidos por el algoritmo genético reducen significativamente el tamaño del conjunto trivial con respecto a los de las otras propuestas en la mayoría de los casos. Sin embargo, para el patrón *Corner tree* con una instancia de 23×23, no logra superar la reducción de la propuesta PS-BB. Por este motivo, el objetivo de la siguiente etapa de experimentaciones consistió en encontrar una configuración de parámetros que redujera el conjunto de mosaicos del patrón *Corner tree* y que mantenga o supere la reducción que se muestra en la Tabla 1 con el resto de los patrones.

4.4. Configuración de parámetros

Una de las principales desventajas de los algoritmos genéticos es la configuración de parámetros, dado que requiere ajustar un conjunto extenso de ellos. La generación de la población está altamente ligada con los resultados posibles a obtener, ya que entre más aleatoriedad exista entre los individuos de la población, se explora mayor campo de soluciones del problema y más por la probabilidad de encontrar un óptimo global. La representación del individuo es otro aspecto muy importante para los algoritmos genéticos, el individuo tiene que representar las características que se desean evolucionar, que solucionen el problema dado (Eiben y Smith, 2003). Como se menciona en la sección anterior, el conjunto de parámetros en el algoritmo genético para el problema PATS consta de seis variables que requieren ser ajustadas. En la Tabla 2 se muestran los diferentes valores asignados al conjunto de parámetros durante las pruebas.

Tabla 2: Parámetros del algoritmo genético para problema PATS.

Parámetros	Opciones
Tamaño de población	50
Número de generaciones	150,000
Probabilidad de cruce	0.3, 0.6 y 0.8
Probabilidad de mutación	0.1, 0.3 y 0.5
Tipo de cruce	horizontal, vertical y marco
Tipo de mutación	color, fila y máscara

En el caso de las probabilidades para cruce y mutación, por convención se utilizó primero una probabilidad de 0.6 para el cruce y 0.1 para la mutación. Sin embargo, durante la experimentación se probaron diferentes valores, arrojando resultados novedosos que se presentan más adelante.

La experimentación del algoritmo genético se realizó utilizando las diferentes combinaciones con los operadores genéticos y los parámetros de probabilidad de cruce y mutación. Es decir, en total se probaron 81 ($= 3 * 3 * 3 * 3$) diferentes combinaciones, cada una de ellas ejecutada 10 veces, en los 16 casos de prueba, lo que da un total de 12,960 ($= 81 * 10 * 16$) ejecuciones. Los parámetros de tamaño de población y número de generaciones no varían su valor asignado.

4.5. Resultados

En esta etapa de la experimentación el objetivo fue encontrar una configuración de parámetros mejor que la de la fase preliminar. La configuración de parámetros que cumple con los requisitos anteriormente descritos fue la siguiente: una población de 50 individuos, aplicándoles 150,000 generaciones con probabilidad de cruce y mutación de 0.8 y 0.5, respectivamente, utilizando el operador de cruce por marco y el de mutación por máscara. En la Tabla 3 se muestra la comparación de los mejores individuos obtenidos con la configuración preliminar y la configuración seleccionada de la experimentación del AG-PATS, para cada caso de prueba. Claramente se puede observar que la configuración seleccionada tiene un mejor desempeño en casi todos los casos de prueba que la configuración preliminar. De los pocos casos de prueba en que la configuración seleccionada

no reduce el número de tipos de mosaicos en el conjunto, es en el patrón *Binary counter* de 6×6, dado que los cuatro mosaicos encontrados constituyen el conjunto óptimo para esta instancia. El otro caso es el de Sierpinski de 6×6 con 7 mosaicos, en donde la reducción obtenida no alcanzó al conjunto óptimo (4 mosaicos).

Para obtener el porcentaje de reducción del conjunto de tipos de mosaicos, se usa la regla de tres. Es decir, se toma el número de mosaicos del conjunto reducido y se multiplica por cien, el resultado se divide con el número de mosaicos del conjunto trivial y se obtiene un porcentaje como resultado. A este resultado se le resta el cien por ciento y así se obtiene el porcentaje de reducción en los conjuntos de cada caso de prueba que se muestra en la Tabla 3.

Tabla 3: Tabla comparativa del mejor individuo obtenido de la configuración preliminar y la configuración seleccionada para el AG-PATS. En negritas se identifica la mejor solución.

Patrón	Tamaño	Conjunto trivial	Conf. preliminar	Conf. seleccionada
Binary counter	6×6	36	**(88.9 %)4**	**(88.9 %)4**
	15×15	225	(92.5 %)17	**(97.7 %)5**
	32×32	1024	(98 %)20	**(98.1 %)19**
Corner tree	23×23	529	(93.4 %)35	**(97.9 %)11**
Chess board	6×6	36	(75 %) 9	**(94.4 %)2**
	15×15	225	(85.5 %)32	**(95.1 %)11**
Random	12×12	144	(86 %)19	**(93 %)10**
	16×16	256	(91 %)23	**(93.3 %)17**
	20×20	400	(91.8 %)33	**(94.7 %)21**
	32×32	1024	(93.3 %) 69	**(96.6 %)31**
Sierpinski	6×6	36	**(80.5 %)7**	**(80.5 %)7**
	8×8	64	(79.7 %)13	**(85.9 %)9**
	12×12	144	(86.2 %)20	**(87.5 %)18**
	17×17	289	(92.8 %)21	**(93 %)20**
	32×32	1024	(96.1 %)40	**(97.7 %)23**
	64×64	4096	(98 %)82	**(99.3 %)25**

La Tabla 4 presenta el promedio y desviación estándar para cada caso de prueba. El promedio se obtiene de 10 ejecuciones del AG-PATS usando la configuración de paráme-

tros seleccionada, esto para cada caso de prueba. También se presenta para fines de comparación el mejor individuo obtenido, así como el tiempo de procesamiento computacional para cada uno de ellos.

Tabla 4: Resultados estadísticos del desempeño del AG-PATS con los diferentes casos de prueba.

Patrón	Tamaño	Promedio	Desviación estándar	Mejor individuo	Tiempo seg.
Binary	6×6	13	$\pm$1.446	4	22
counter	15×15	12.8	$\pm$2.664	5	105
	32×32	28	$\pm$3.429	19	650.
Corner tree	23×23	24.25	$\pm$4.456	11	401
Chess	6×6	11	$\pm$2.375	2	30
board	15×15	29	$\pm$1.504	11	108
Random	12×12	23	$\pm$10.263	10	67
	16×16	28	$\pm$7.008	17	127
	20×20	39.7	$\pm$17.820	21	232
	32×32	39.14	$\pm$10.286	31	802
Sierpinski	6×6	12.5	$\pm$4.006	7	22
	8×8	15.9	$\pm$6.910	9	44
	12×12	31.3	$\pm$8.056	18	101
	17×17	30.6	$\pm$11.047	20	288
	32×32	49.4	$\pm$24.357	23	1096
	64×64	94.7	$\pm$57.956	25	4872

De acuerdo a estos resultados, se puede observar que para el patrón *Binary counter* con sus distintas instancias (Figuras 27, 28 y 29), el AG-PATS obtiene una desviación estándar relativamente pequeña (± 3.5 en el caso más grande). Es deseable que la desviación estándar sea lo más reducida posible, ya que de ser así, se podría asegurar que los resultados obtenidos sean altamente similares a la media con pocas experimentaciones. En la Figura 27 se presenta la evolución del individuo promedio utilizando el patrón *Binary counter* con instancia de 32×32. Aquí, la curva de desviación estándar es muy cercana al individuo promedio, lo cual nos dice que el algoritmo converge a soluciones con reducciones similares. Sin embargo, para los patrones *Random* y Sierpinski, el algo-

ritmo obtiene una desviación estándar muy elevada, como se muestra gráficamente en las figuras 34, 35, 36, 40, 41 y 42; donde la curva de la desviación estándar se aprecia con una separación considerable con respecto a la curva del individuo promedio.

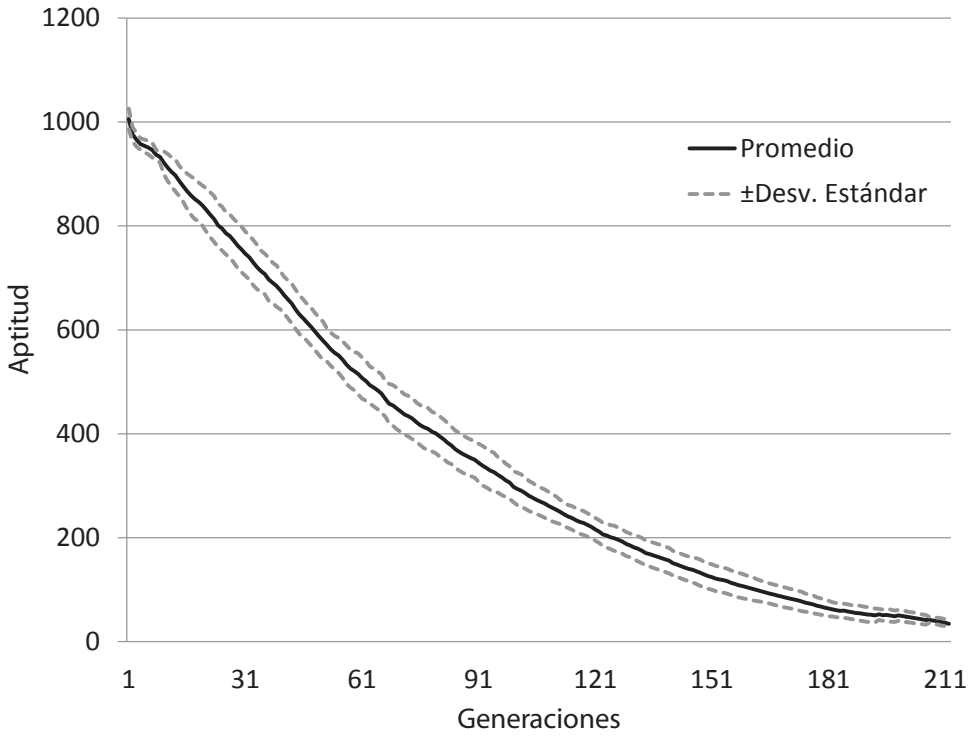


Figura 27: Individuo promedio contra número de generaciones para el patrón *Binary Counter* (instancia de 32×32).

Realizando una comparación entre el individuo promedio y el mejor individuo, observamos que para los casos de prueba de *Binary counter* de 15×15, *Corner tree* de 23×23, *Chess board* de 15×15, *Random* de 12×12, Sierpinski de 12×12 y 32×32, el individuo promedio duplica el número de mosaicos en el conjunto en relación con el mejor individuo. Cabe destacar que para el patrón *Binary counter* de 6×6 el promedio es 3 veces más grande y para Sierpinski 64×64 y *Chess board* de 6×6 es 4 y 5 veces más grande que el mejor individuo de sus respectivos casos de prueba. Sin embargo, la diferencia de mosaicos entre el promedio y el mejor individuo para *Binary counter* de 32×32 y *Random* de 32×32 es de 9 y 8 mosaicos, respectivamente; es decir, el mejor individuo de estos últimos dos patrones es más cercano al promedio. Por lo que se puede concluir, que el AG-PATS tiene un mejor desempeño en los casos de prueba con instancias grandes que con instancias medianas y pequeñas.

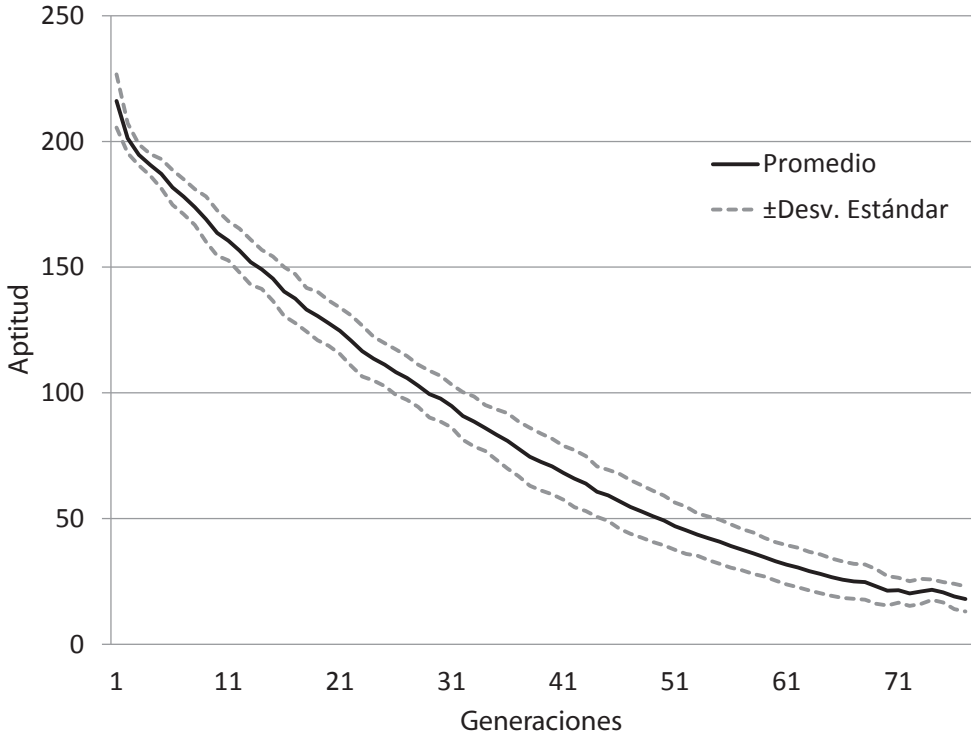


Figura 28: Individuo promedio contra número de generaciones para el patrón *Binary Counter* (instancia de 15×15).

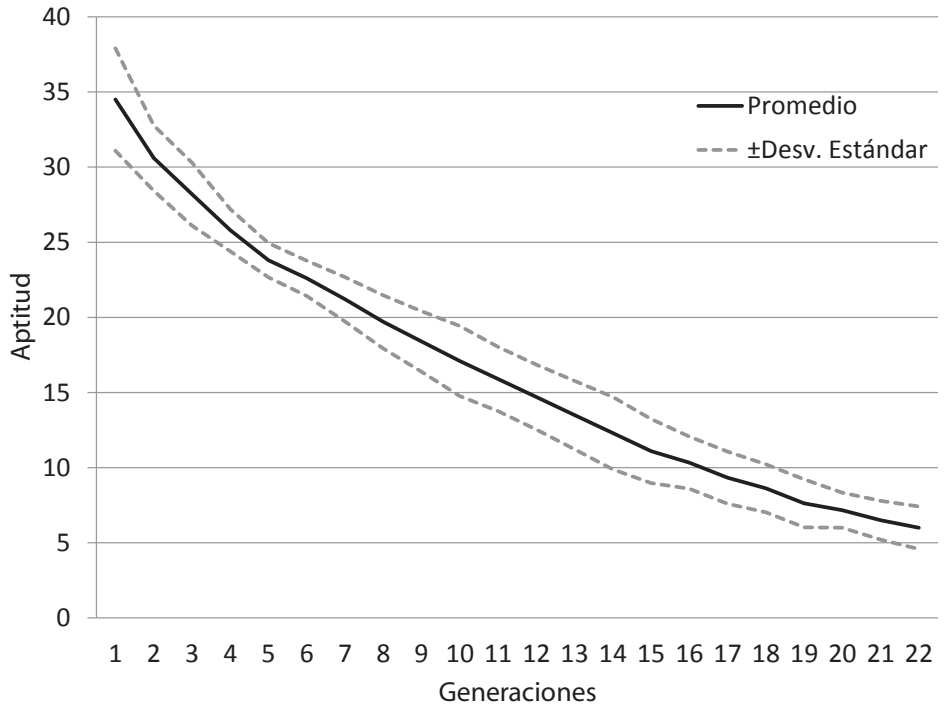


Figura 29: Individuo promedio contra número de generaciones para el patrón *Binary Counter* (instancia de 6×6).

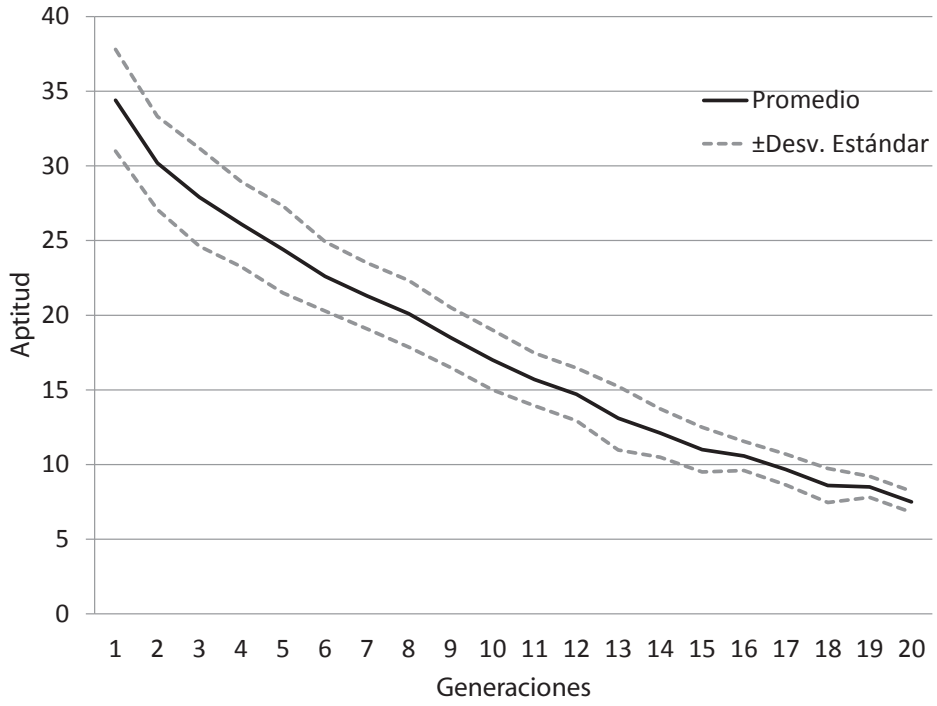


Figura 30: Individuo promedio contra número de generaciones para el patrón *Chess board* (instancia de 6×6).

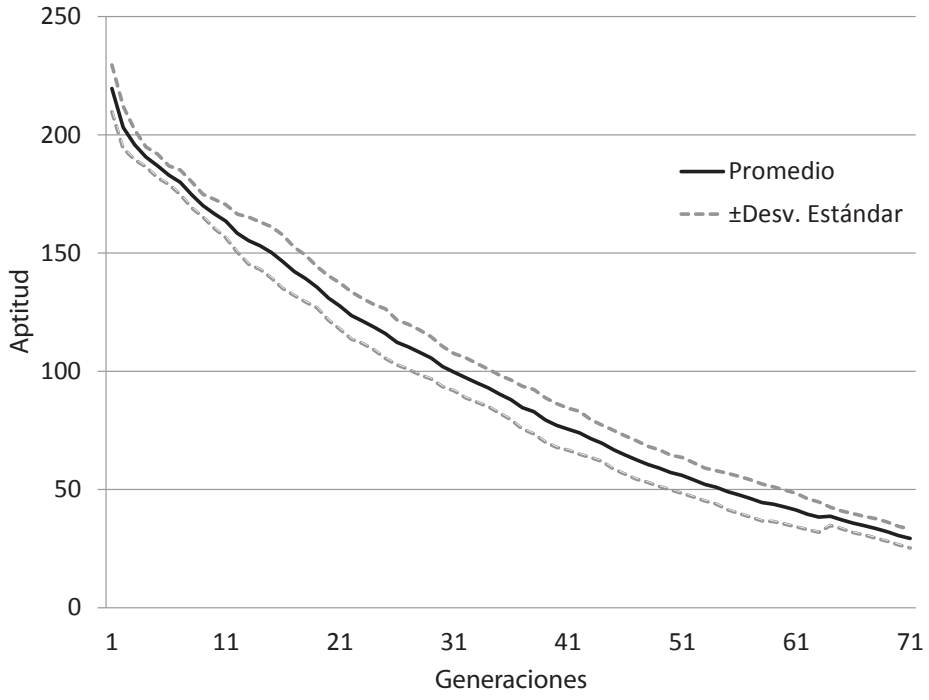


Figura 31: Individuo promedio contra número de generaciones para el patrón *Chess board* (instancia de 15×15).

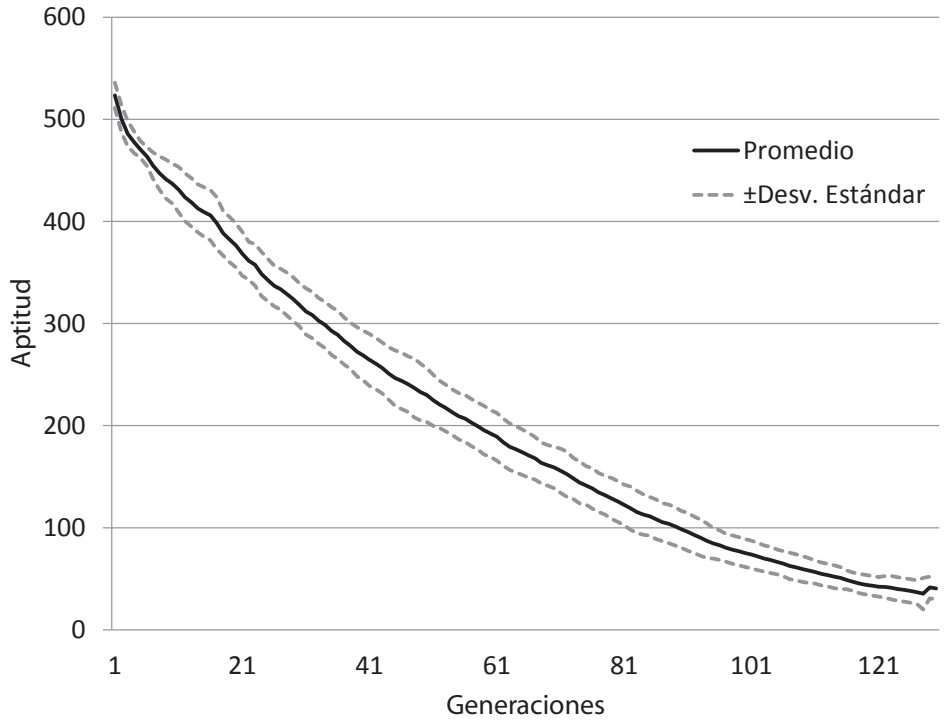


Figura 32: Individuo promedio contra número de generaciones para el patrón *Corner Tree* (instancia de 23×23).

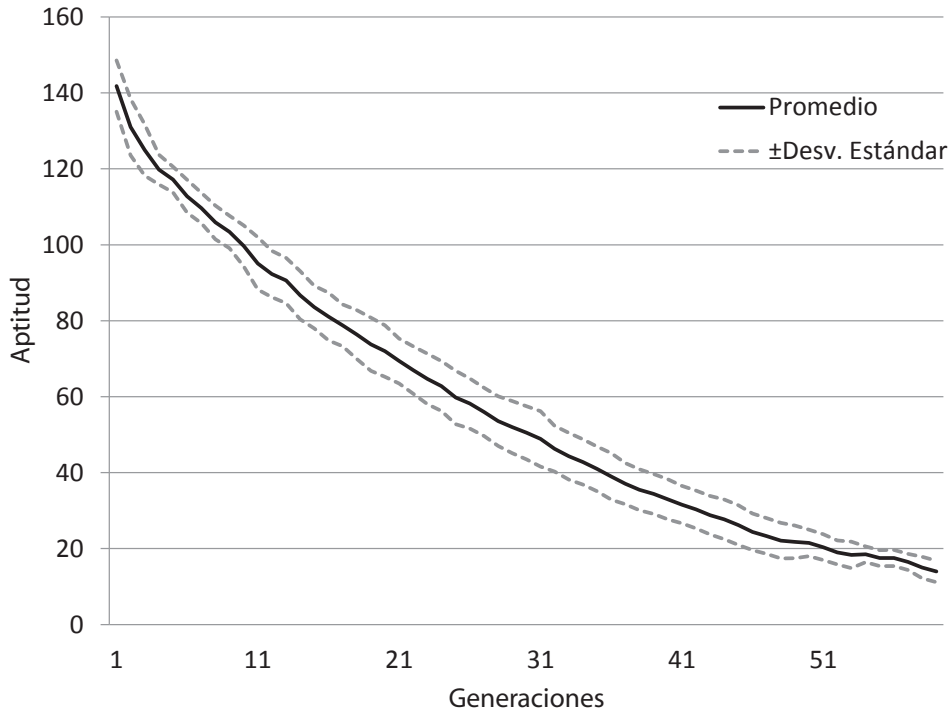


Figura 33: Individuo promedio contra número de generaciones para el patrón *Random* (instancia de 12×12).

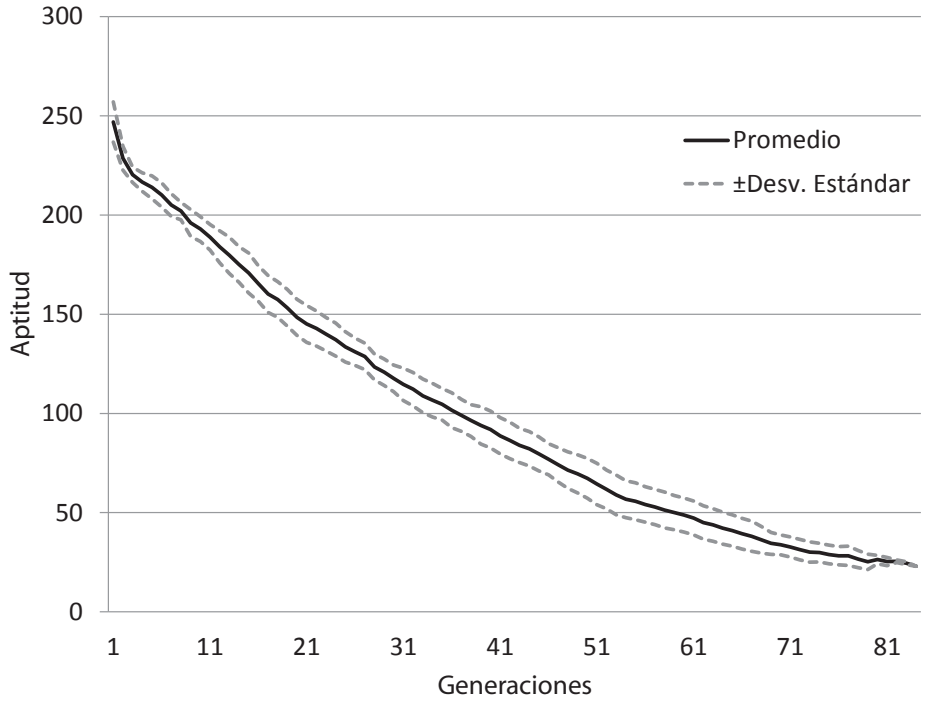


Figura 34: Individuo promedio contra número de generaciones para el patrón *Random* (instancia de 16×16).

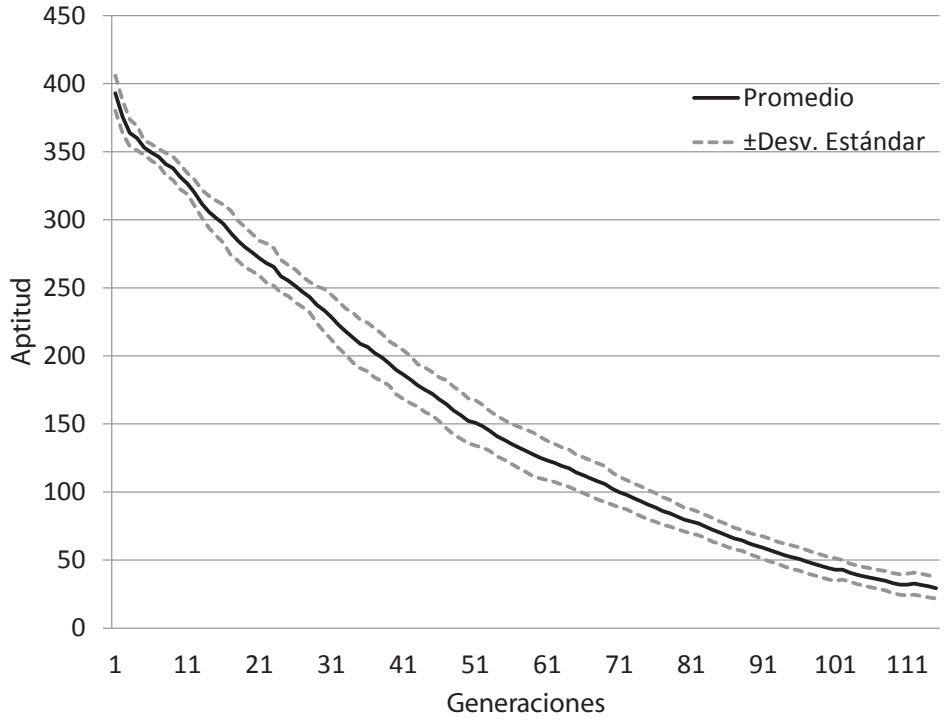


Figura 35: Individuo promedio contra número de generaciones para el patrón *Random* (instancia de 20×20).

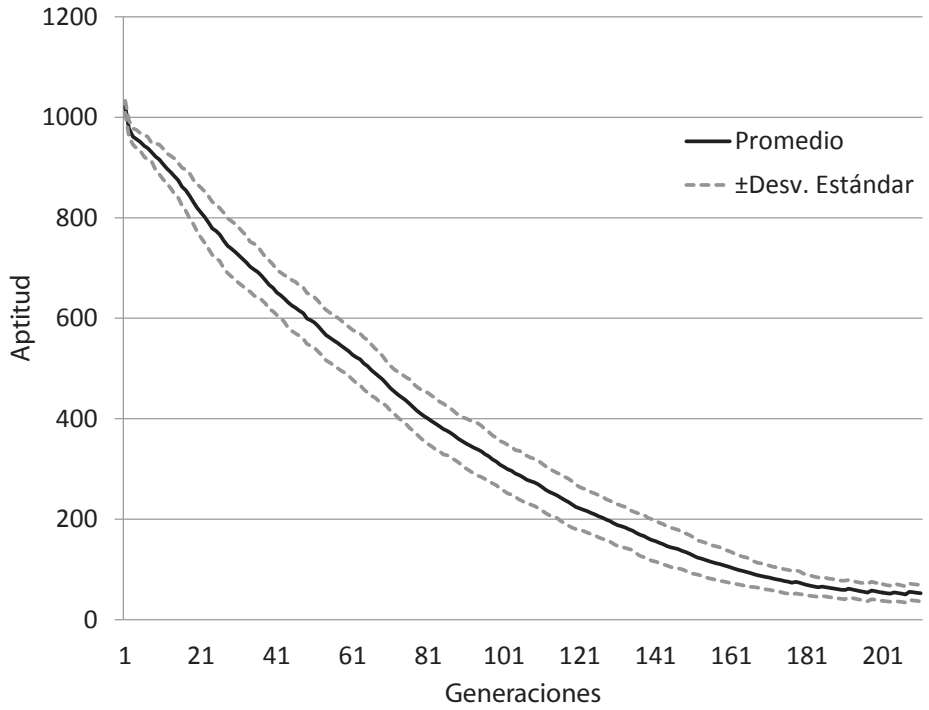


Figura 36: Individuo promedio contra número de generaciones para el patrón *Random* (instancia de 32×32).

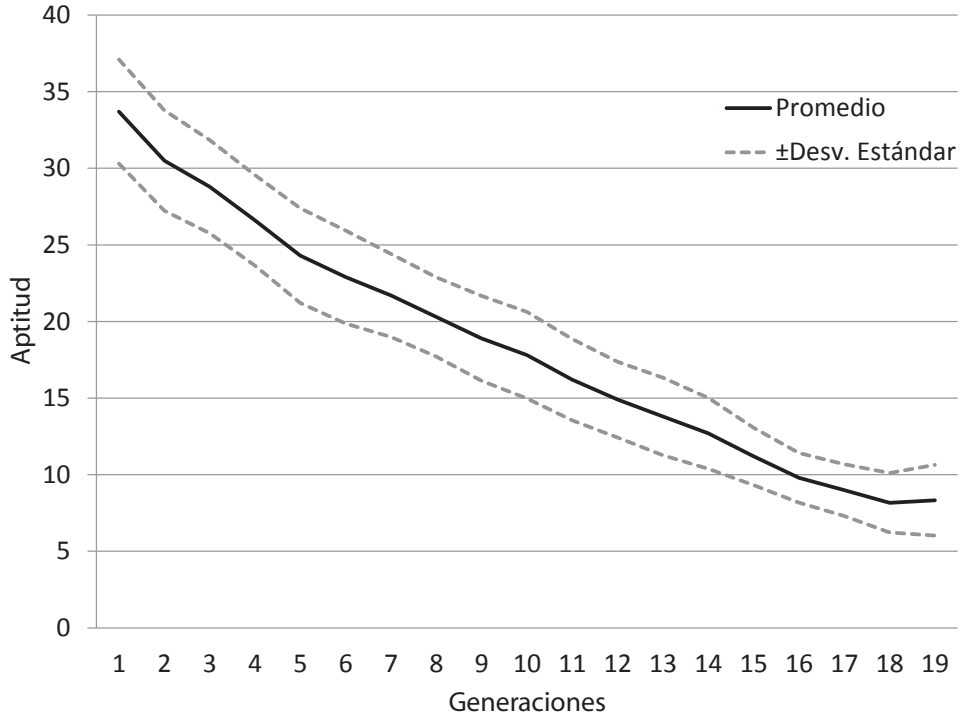


Figura 37: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 6×6).

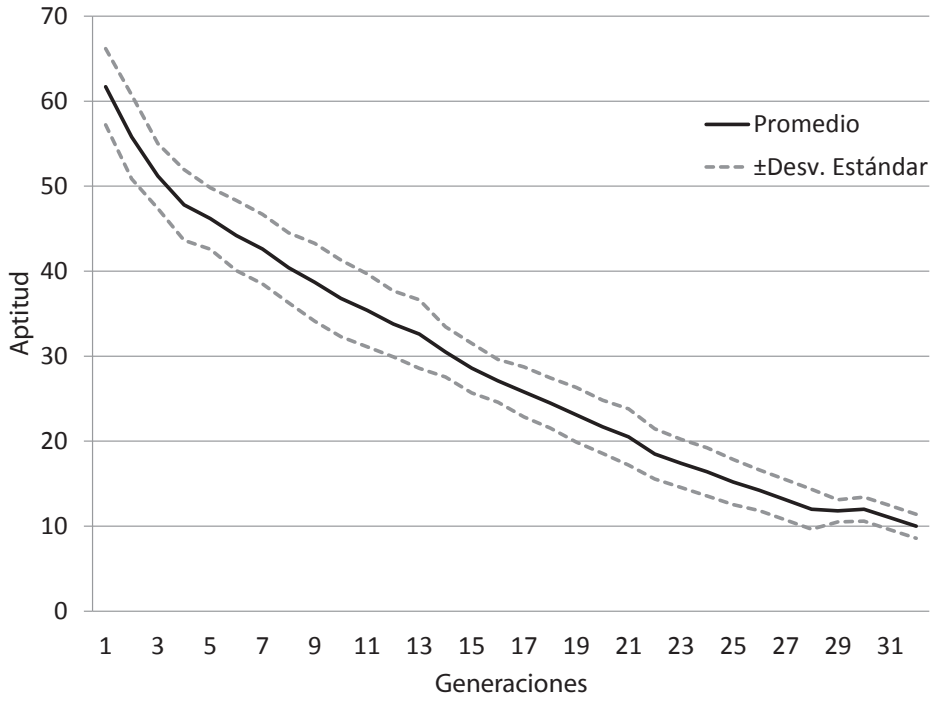


Figura 38: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 8×8).

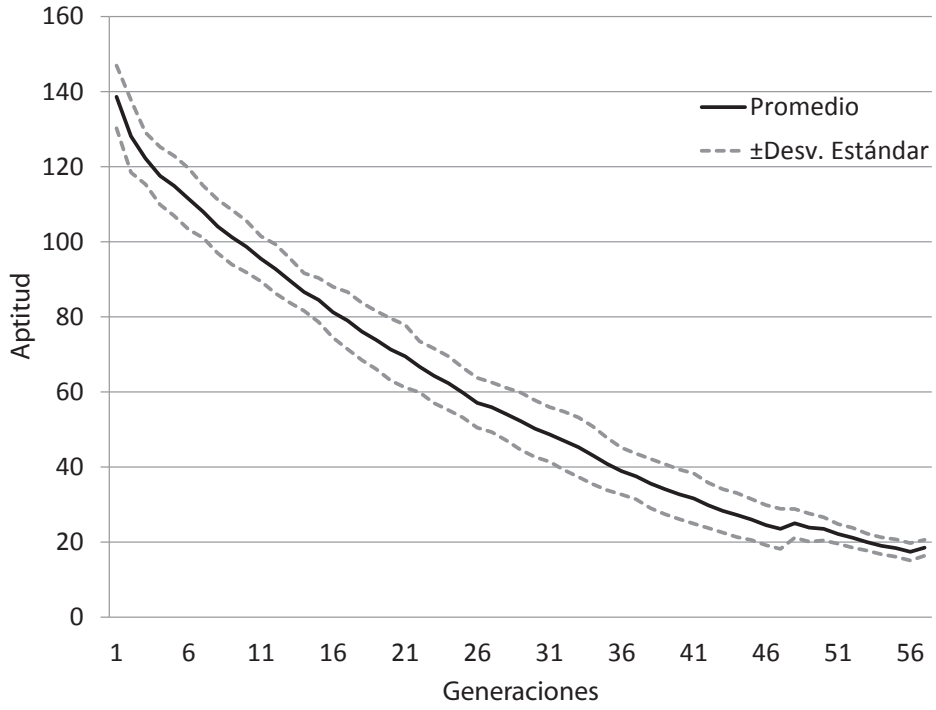


Figura 39: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 12×12).

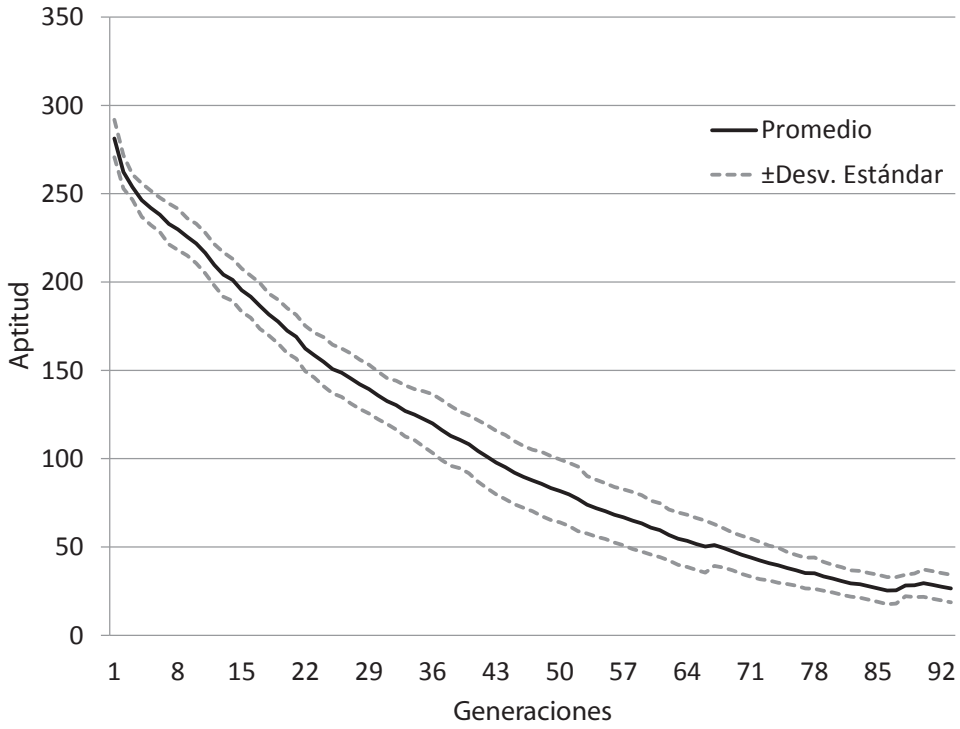


Figura 40: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 17×17).

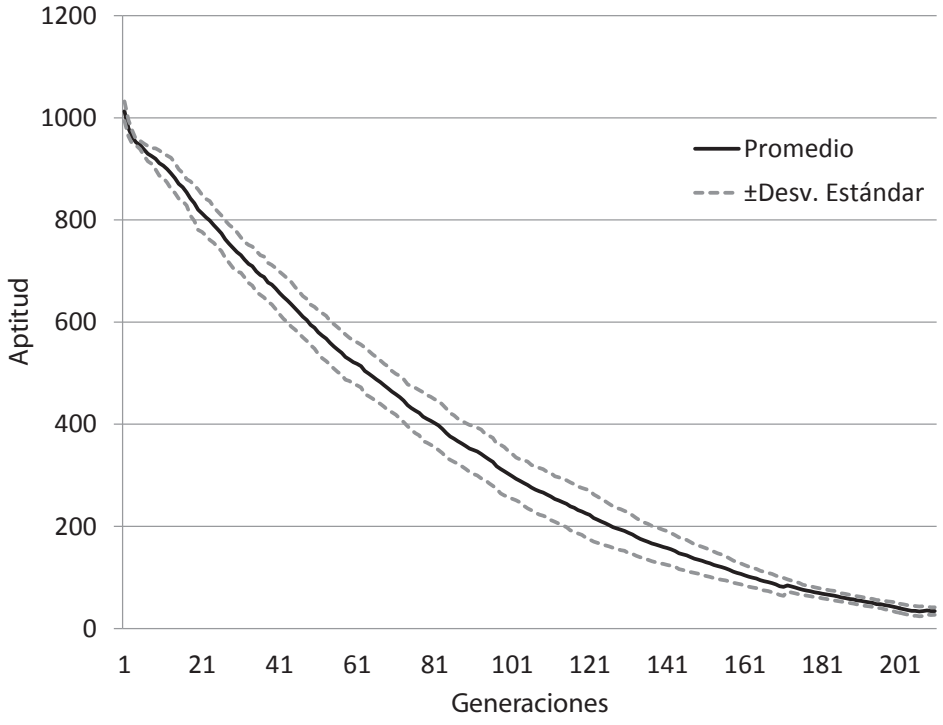


Figura 41: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 32×32).

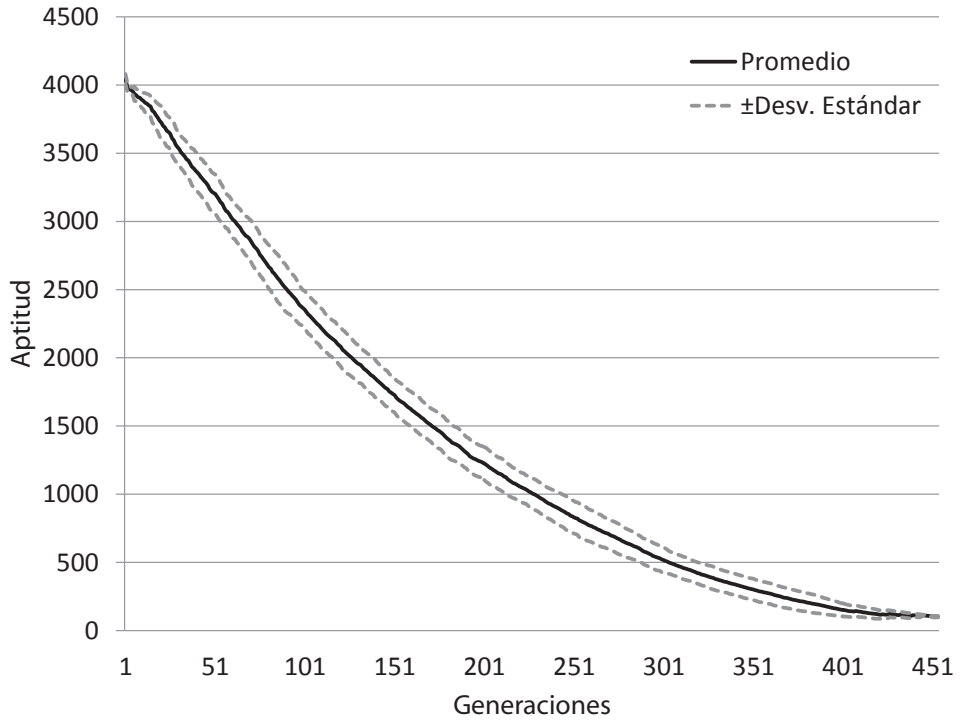


Figura 42: Individuo promedio contra número de generaciones para el patrón Sierpinski (instancia de 64×64).

Tabla 5: Tabla comparativa de resultados del AG-PATS con las propuestas del estado del arte: [1] Ma y Lombardi (2008), [2] Göös y Orponen (2010) y [3] Lempiäinen *et al.* (2011). En negritas se identifica la mejor solución.

Patrón	Conjunto trivial	Voraz [1]	B&B [2]	PS-BB [3]	AG-PATS Promedio	AG-PATS Mejor
Binary	(6×6) 36	(47 %)19	$n \setminus a$	$n \setminus a$	(63.9 %)13	**(88.9 %)4**
counter	(15×15)225	(46 %)121	$n \setminus a$	$n \setminus a$	(87.6 %)28	**(97.7 %)5**
	(32×32)1024	$n \setminus a$	(93.9 %)62	(98 %)20	(98.3 %)12	**(98.1 %)19**
Corner	(23×23)529	$n \setminus a$	$n \setminus a$	(95.2 %)25	(95.5 %)24	**(97.9 %)11**
tree						
Chess	(6×6) 36	(47 %)19	$n \setminus a$	$n \setminus a$	(69.5 %)11	**(94.4 %)2**
board	(15×15)225	(47 %)119	$n \setminus a$	$n \setminus a$	(87.2 %)29	**(95.1 %)11**
Random	(12×12)144	$n \setminus a$	(59.7 %)58	$n \setminus a$	(84.1 %)23	**(93 %)10**
	(16×16)256	$n \setminus a$	(60.9 %)100	$n \setminus a$	(89.1 %)28	**(93.3 %)17**
	(20×20)400	$n \setminus a$	(60 %)160	$n \setminus a$	(90.2 %)39	**(94.7 %)21**
	(32×32)1024	$n \setminus a$	(58.9 %)420	$n \setminus a$	(96.2 %)39	**(96.6 %)31**
Sierpinski	(6×6) 36	(42 %)21	$n \setminus a$	$n \setminus a$	(67 %)12	**(80.5 %)7**
	(8×8) 64	(43 %)37	$n \setminus a$	$n \setminus a$	(76.6 %)15	**(85.9 %)9**
	(12×12)144	(45 %)80	(86.9 %)20	$n \setminus a$	(78.5 %)31	**(87.5 %)18**
	(17×17)289	(41 %)172	(92 %)23	$n \setminus a$	(89.7 %)30	**(93 %)20**
	(32×32)1024	(42 %)595	(94.1 %)60	(96 %)42	(95.3 %)49	**(97.7 %)23**
	(64×64)4096	$n \setminus a$	$n \setminus a$	(97 %)95	(97.8 %)94	**(99.3 %)25**

La evolución del individuo promedio con su respectiva desviación estándar utilizó 150,000 generaciones para cada caso de prueba, sin embargo, por cuestión de análisis visual se delimitaron las figuras en el número de generación que el individuo promedio ya no mostraba cambio en la reducción.

El mejor individuo de la configuración de parámetros seleccionada fue también comparado con los resultados del estado del arte para cada caso de prueba. Dado que para los algoritmos probabilísticos es importante el promedio de las experimentaciones, también se incluye el individuo promedio en la tabla comparativa. La Tabla 5 muestra el conjunto de tipos de mosaicos obtenidos por las diferentes propuestas. De igual forma, se presenta el porcentaje de reducción en los mismos. Para el patrón de Sierpinski con instancia de 32×32 se puede observar la reducción del conjuntos de tipos de mosaicos en cada una de las propuestas, en donde el algoritmo voraz reduce un 42 % el conjunto de tipos de mosaicos con respecto al conjunto trivial. La propuesta de B&B lo reduce en un 94.1 % con un tiempo de procesamiento computacional de dos horas y el PS-BB alcanza una reducción de 96 % con un tiempo de procesamiento computacional de treinta segundos, no obstante, el mejor individuo obtenido del AG-PATS lo reduce en un 97.7 % (Figura 43)con un tiempo de procesamiento computacional de veinte minutos. Sin embargo, la reducción del individuo promedio se queda con un 95.3 %.

En el caso del patrón *Corner tree* con instancia de 23×23, observamos que la comparación únicamente se presenta con la propuesta de PS-BB y AG-PATS (mejor individuo), ya que las otras heurísticas no realizaron dicho experimento. En la Figura 44 se aprecia gráficamente la reducción significativa que alcanza el AG-PATS con respecto al conjunto trivial, reduce el doble de mosaicos que el algoritmo PS-BB. Para el patrón de *Binary counter* con instancia de 32×32 la comparación se realiza entre las propuestas de B&B, PS-BB y AG-PATS. La Figura 45 muestra la reducción del conjunto de tipos de mosaicos que obtiene cada una de las propuestas. El resultado de PS-BB y el mejor individuo del AG-PATS se diferencia por un mosaico: El PS-BB encuentra 20 tipos de mosaicos y un porcentaje de reducción de 98 %, mientras que el AG-PATS reduce a 19 tipos de mosaicos con un porcentaje de reducción del 98.1 %. El B&B se queda con 62 tipos de mosaicos y un porcentaje de 93.9 %.

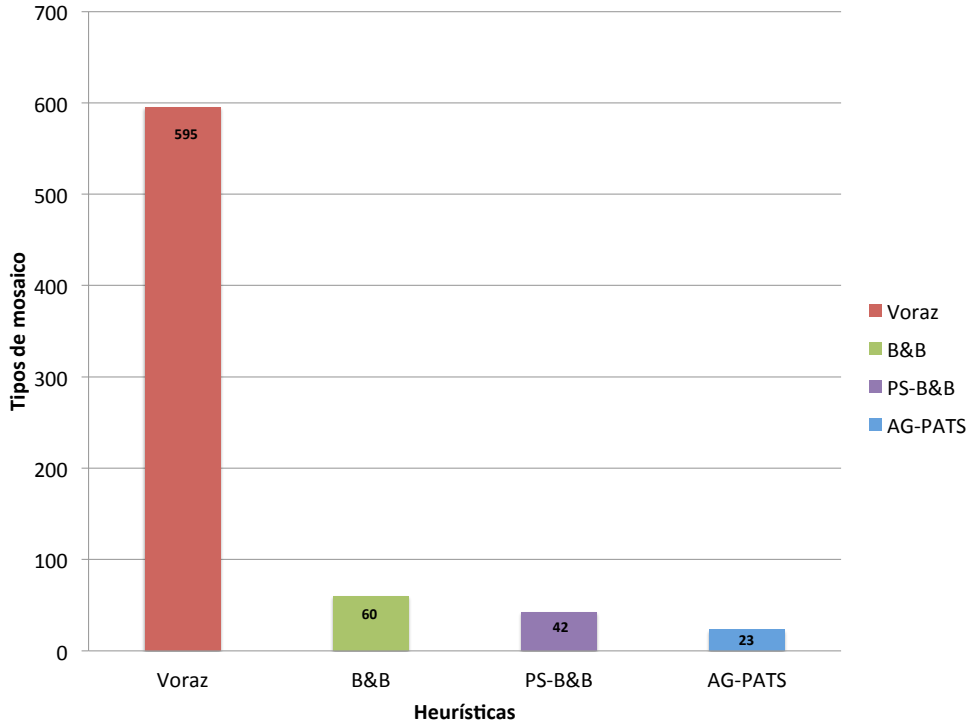


Figura 43: Comparación del conjunto de tipos de mosaicos para el patrón Sierpinski de 32×32. [Voraz] Ma y Lombardi (2008), [B&B] Göös y Orponen (2010) y [PS-BB] Lempiäinen *et al.* (2011)

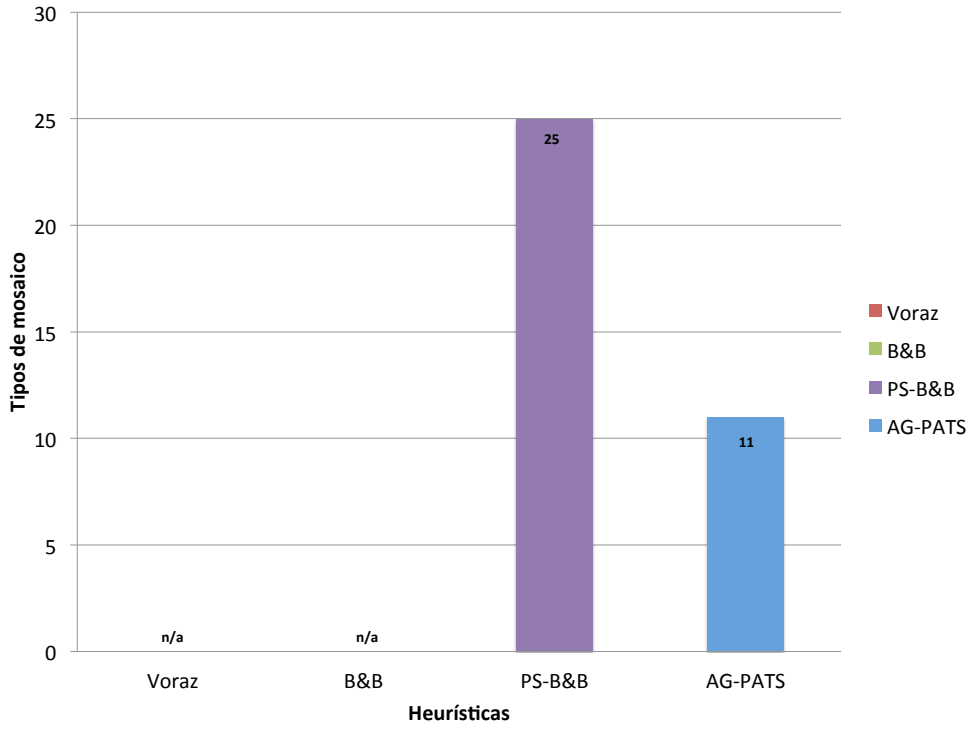


Figura 44: Comparación del conjunto de tipos de mosaicos para el patrón *Corner Tree* de 23×23. [Voraz] Ma y Lombardi (2008), [B&B] Göös y Orponen (2010) y [PS-BB] Lempiäinen *et al.* (2011)

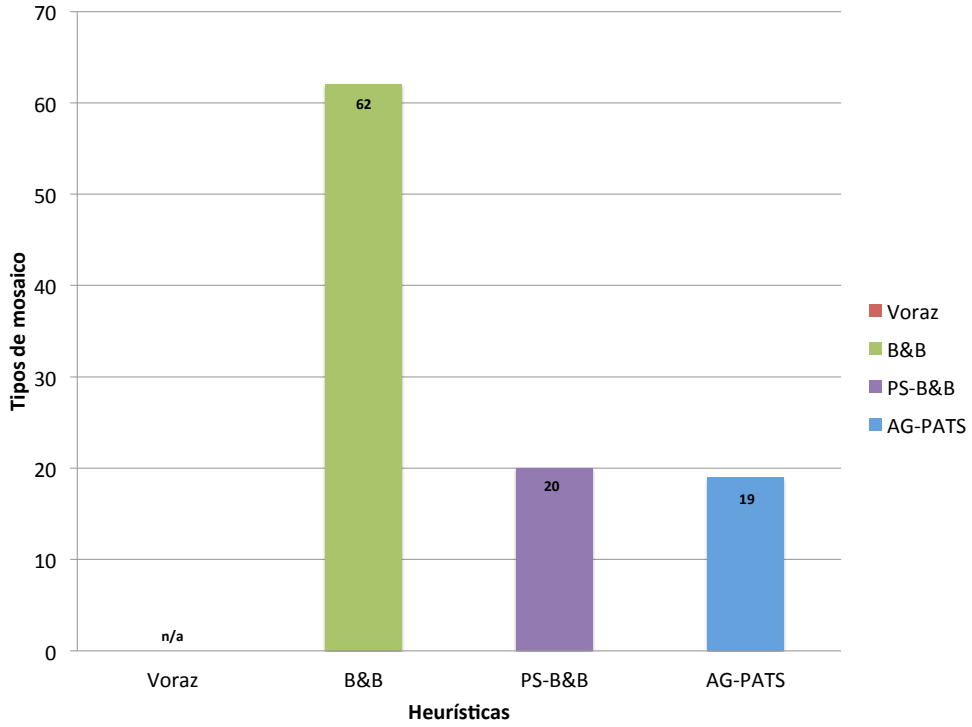


Figura 45: Comparación del conjunto de tipos de mosaicos para el patrón *Binary counter* de 32×32. [Voraz] Ma y Lombardi (2008), [B&B] Göös y Orponen (2010) y [PS-BB] Lempiäinen *et al.* (2011)

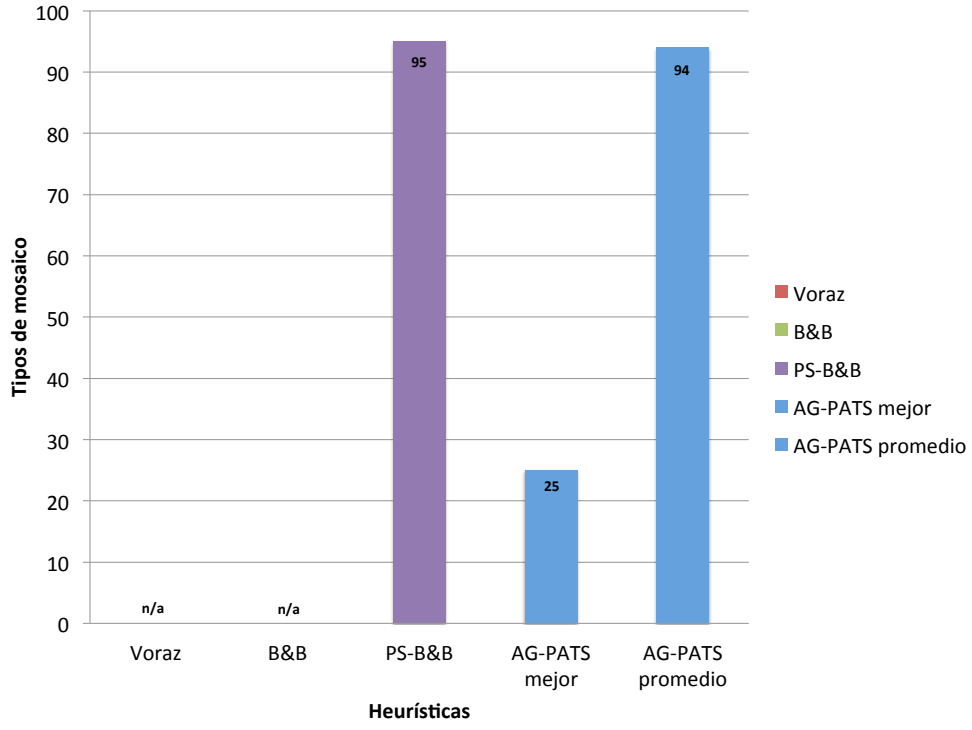


Figura 46: Comparación del conjunto de tipos de mosaicos para el patrón Sierpinski de 64×64. [Voraz] Ma y Lombardi (2008), [B&B] Göös y Orponen (2010) y [PS-BB] Lempiäinen *et al.* (2011)

Para el caso de prueba más grande, el patrón de Sierpinski de 64×64, la comparación se realiza entre el algoritmo PS-BB y AG-PATS. En la Figura 46 se observa que la heurística PS-BB reduce el conjunto trivial hasta llegar a 95 mosaicos, mientras que AG-PATS lo reduce a 25 mosaicos con el mejor individuo y con el individuo promedio lo reduce a 94 mosaicos. El AG-PATS supera la reducción de PS-BB incluso con el individuo promedio. En general el desempeño del AG-PATS realiza una reducción considerable en cada caso de prueba. Sin embargo, existe oportunidad de mejora, dado que no se obtuvo el conjunto óptimo en algunos casos donde éste se conoce. Por ejemplo para el patrón de Sierpinski y *Binary counter*, el conjunto mínimo es de 4 tipos de mosaicos. Para *Chess board* el mínimo es de 2 mosaicos. En el caso del patrón *Random* y *Corner tree* se desconoce su conjunto óptimo.

Con respecto al tiempo de procesamiento computacional, el AG-PATS converge en 90 minutos. Por ejemplo, para el patrón de Sierpinski con la instancia más grande de 64×64 se tarda una hora y media en reducir el conjunto trivial a 25 tipos de mosaicos (mejor individuo) y para la instancia de 32×32 se tarda 20 minutos en reducir a 23 mosaicos en el conjunto trivial. B&B es un algoritmo que garantiza el óptimo explorando el espacio completo de búsqueda, por lo que el tiempo máximo asignado de procesamiento fue de 2 horas para cada caso de prueba, con un procesador AMD de 2.61 GHz. En el caso de PS-BB, el procesamiento computacional se realizó de forma paralela, y se reporta que para el patrón de Sierpinski de 32×32 se tarda 30 segundos en reducir el conjunto trivial a 42 tipos de mosaicos mientras que para la instancia de 64×64, se tarda una hora en reducir a 95 tipos de mosaicos. De esta manera, comparando entre AG-PATS y PS-BB en relación a el tiempo de procesamiento y reducción obtenida para el patrón de Sierpinski de 64×64, el AG-PATS se tarda media hora más que PS-BB. Sin embargo, este último reduce el conjunto de tipos de mosaicos en un 97 %, mientras que el AG-PATS lo reduce en un 99.3 %.

Los conjuntos de mosaicos obtenidos por el AG-PATS del mejor individuo para cada caso de prueba, se muestran en el Apéndice A.

4.5.1. Simulación en ISU TAS

El software ISU TAS provee simulaciones gráficas del auto-ensamblamiento de mosaicos. En esta sección se presenta la simulación del caso de prueba Sierpinski de 8×8. El proceso de simulación inicia con la traducción de formato del conjunto de mosaicos, donde se observa que la herramienta ISU TAS utiliza más características en el diseño de mosaicos que el AG-PATS.

```
BEGIN GLUEFUNCTIONS
[0
  1: zero
]
[1
  1: one
]
[2
  2: TWO
]
[TWO
  2: 2
  2: TWO
]
[one
  1: 1
  1: one
]
[zero
  1: 0
  1: zero
]
END GLUEFUNCTIONS
```

(a)

```
TILENAME SEED
LABEL
NORTHBIND 0
EASTBIND 0
SOUTHBIND 0
WESTBIND 0
NORTHLABEL 2
EASTLABEL 2
SOUTHLABEL
WESTLABEL
TILECOLOR red
```

```
64 64 2
C N S E O
1 2 1 1 1
0 3 1 2 2
0 4 1 3 2
0 5 1 4 2
0 6 1 5 2
0 7 1 6 2
0 8 1 7 2
0 9 1 8 2
1 1 1 9 2
1 2 2 1 2
0 3 2 2 3
0 4 2 3 3
0 5 2 4 3
```

(b)

Figura 47: Formatos del diseño de mosaicos: (a) ISU TAS y (b) AG-PATS.

Por ejemplo, el AG-PATS utiliza color y fuerzas de pegado en el norte, sur, este y oeste (Figura 47 (b)); en cambio ISU TAS utiliza fuerzas de pegado para cada orientación y adicionalmente usa etiquetas para el norte, sur, este y oeste (Figura 47 (a)). Para realizar la traducción de formatos en este caso específico, se tomó como plantilla un ejemplo de ISU TAS. La función que determina las fuerzas de pegado se ajustó al rango que maneja el AG-PATS y se estructuraron los mosaicos en base a dicha plantilla. Posteriormente, se ejecutó el software y se realizó la simulación, utilizando la opción del modelo de auto-ensamblado aTAM. En la Figura 48 se muestra el diseño del conjunto de mosaicos y la simulación de auto-ensamblamiento para esta instancia.

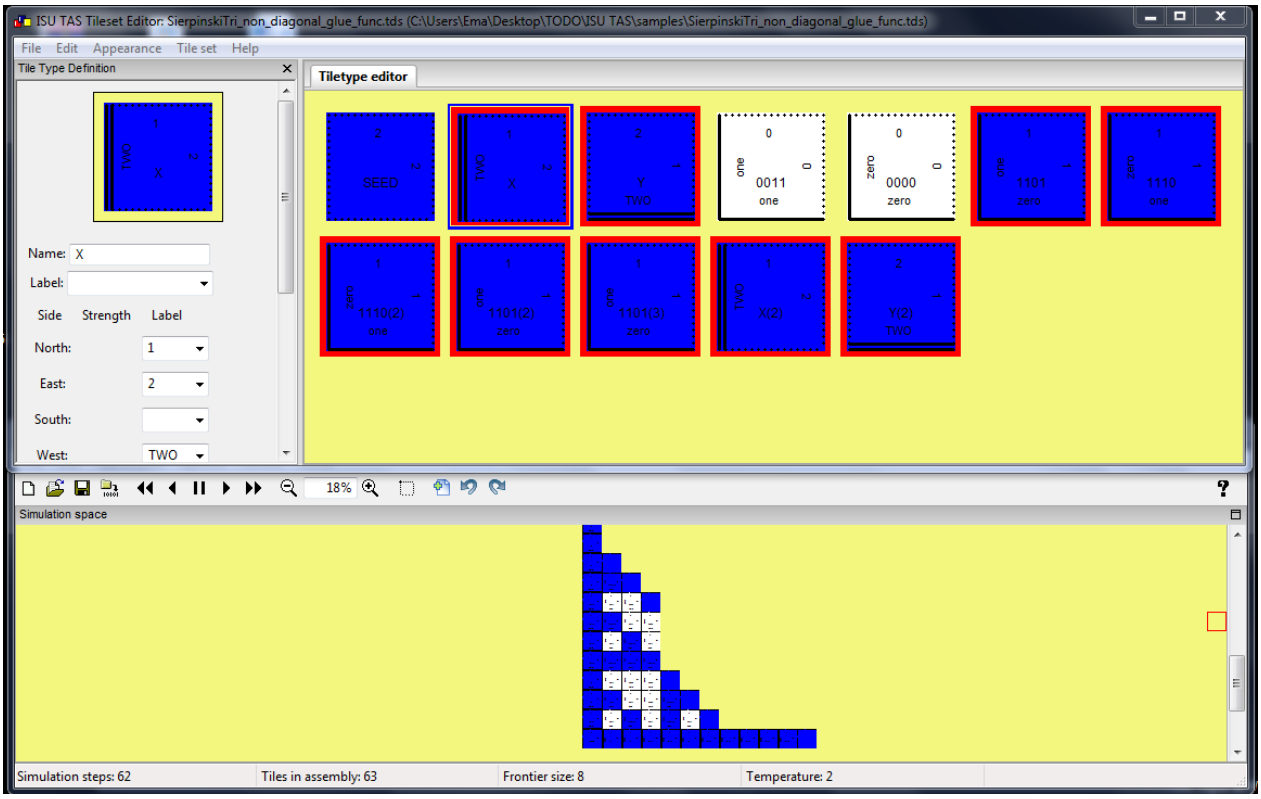


Figura 48: Simulación del mejor individuo del patrón Sierpinski con instancia de 8×8 en ISU TAS.

Capítulo 5. Conclusiones y trabajo futuro

5.1. Sumario

En este trabajo se abordó el problema "*Patterned self-Assembly Tile set Synthesis*" (PATS) del modelo de auto-ensamblado de ADN. El problema consiste en encontrar un conjunto mínimo de tipos de mosaicos que sean capaces de formar un patrón P dado, mediante auto-ensamblado de moléculas de ADN. Se desarrolló un algoritmo genético que encuentra soluciones comparables con las logradas por métodos del estado del arte. En el algoritmo genético para el problema PATS (AG-PATS), se diseñó la representación del individuo, inicialización de la población, operadores genéticos y el mecanismo de evaluación de la función objetivo. Los casos de prueba para este problema se obtuvieron de la literatura. Se realizó una experimentación con el objetivo de encontrar un conjunto de parámetros para que el algoritmo obtuviera soluciones. La configuración de parámetros seleccionada alcanza soluciones que superan los resultados de la literatura. Sin embargo, como los algoritmos genéticos son probabilísticos, el hecho de encontrar una buena solución en una experimentación, no asegura que el algoritmo vuelva a encontrar esa solución en ejecuciones posteriores, por lo que se requiere de datos estadísticos, tales como promedio y desviación estándar. Entre más pequeña sea la desviación estándar, es más probable volver a encontrar soluciones similares.

5.2. Conclusiones

1. Es importante garantizar que dentro de la población inicial se tenga diversidad de soluciones para evitar la convergencia prematura del algoritmo genético. La restricción inicial del problema es que para realizar el auto-ensamblado del patrón (individuo) el conjunto de mosaicos sea trivial. Por lo que la generación de la población inicial se realizó mediante la fabricación de un conjunto trivial diferente para cada individuo, de esta forma se le dio aleatoriedad a la población inicial del AG-PATS.

2. Los operadores genéticos son los encargados de evolucionar al individuo en el algoritmo genético, para esto el individuo debe ser cruzado y posteriormente mutado.

Para el diseño de los operadores de mutación, fue necesario idear una técnica que fuera capaz de realizar un cambio significativo en el individuo. La técnica efectúa la creación de nuevos mosaicos, realizando una combinación de fuerzas de pegado de mosaicos ya existentes para la generación de nuevos, de esta forma se pudo generar operadores de mutación eficaces en la evolución del individuo para el problema PATS.

3. En los resultados se muestra que el AG-PATS supera la reducción en el conjunto de tipos de mosaicos que se presentan en métodos del estado del arte. La propuesta de Ma y Lombardi (2008) es superada significativamente por el algoritmo genético y lo hace para todos los casos de prueba. Cabe destacar además que, el AG-PATS encuentra una solución para cada caso de prueba que supera a los resultados presentados por las propuestas de Göös y Orponen (2010) y Lempiäinen *et al.* (2011).

4. Los algoritmos genéticos usualmente son considerados lentos en el proceso de evolución, sin embargo, el AG-PATS converge a la mejor solución alcanzada para el patrón más grande (Sierpinski 64×64) en 500 generaciones, con un tiempo de procesamiento computacional de una hora y media aproximadamente.

5. El resultado del individuo promedio no fue favorable para la mayor parte de casos de prueba, como se comenta en el Capítulo 4. No obstante, se obtuvo una reducción mayor que las de los métodos del estado del arte para el patrón más grande (Sierpinski 64×64) e incluso, para los casos de prueba con instancias de 32×32 se obtiene un 95 % de reducción. De aquí que se podría inferir que entre más grande sea la instancia, el individuo promedio tiene un mejor desempeño.

6. El problema PATS es relativamente nuevo y por ello pocas estrategias se le han aplicado. Las propuestas existentes en el estado del arte son de tipo determinístico. Y hasta el momento no se había aplicado un algoritmo estocástico, lo cual hace atractivo el presente trabajo y realiza una aportación innovadora al estado del arte.

5.3. Trabajo futuro

1. Dentro del trabajo futuro, el AG-PATS propuesto se puede mejorar en términos de eficiencia de la búsqueda en el espacio de soluciones considerando las distintas

formas de inicialización de la población y combinación de operadores genéticos. Así como la generación de nuevos operadores genéticos y variación en la selección de individuos sobrevivientes.

2. En el aspecto computacional, se puede mejorar la experimentación mediante el uso de hilos de ejecución. En lugar de evolucionar la población secuencialmente se puede paralelizar utilizando un hilo por cada individuo de la población. Esto reduciría el tiempo de procesamiento computacional y abriría nuevas puertas en cuestión de resultados, dado que se podría asignar una configuración de parámetros diferentes para cada individuo, mientras se evolucionan independientemente.

Lista de referencias

Adleman, L. (1996). On Constructing a Molecular Computer. En: *1st DIMACS workshop on DNA based computers*. Vol. 27 de *DIMACS series*, pp. 1–21.

Adleman, L., Cheng, Q., Goel, A., Huang, M.-D., Kempe, D., de Espanés, P. M., y Rothemund, P. W. K. (2002). Combinatorial optimization problems in self-assembly. En: *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, New York, NY, USA. ACM, STOC '02, pp. 23–32.

Adleman, L. M. (1994). Molecular Computation of Solutions to Combinatorial Problems. *Science*, **266**(5187): 1021–1024.

Amos, M. (2005). *Theoretical and experimental DNA computation*, Vol. 4 de *Natural Computing Series*. Springer.

Barua, R. y Misra, J. (2003). Binary arithmetic for DNA computers. *DNA Computing*, pp. 124–132.

Becerril, J. (2013). Ciencia de la vida. Fecha de consulta: 8 de Abril de 2014. URL:http://cienciasvidaccm.blogspot.mx/.

Benenson, Y., Adar, R., Paz-Elizur, T., y Shapiro, E. (2001). Programmable and autonomous computing machine made of biomolecules. *Nature*, **414**(6862): 430–434.

Berger, R. (1966). *The Undecidability of the Domino Problem (No. 66)*. American Mathematical Society.

Cervantes-Salido, V. M., Jaime, O., Brizuela, C. A., y Martínez-Pérez, I. M. (2013). Improving the design of sequences for dna computing: A multiobjective evolutionary approach. *Appl. Soft Comput.*, **13**(12): 4594–4607.

Chambers, L. (1998). *Practical Handbook of Genetic Algorithms: Complex Coding Systems*. Número v. 3. Taylor & Francis.

Chen, H.-L., Cheng, Q., Goel, A., Huang, M.-D., y de Espanés, P. M. (2004). Invadable self-assembly: Combining robustness with efficiency. En: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics, SODA '04, pp. 890–899.

Cormen, T. H., Stein, C., Rivest, R. L., y Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education.

Cuthbert, T. (1987). *Optimization using personal computers: with applications to electrical networks*. A Wiley-Interscience publication. Wiley.

Czeizler, E. y Popa, A. (2012). Synthesizing minimal tile sets for complex patterns in the framework of patterned dna self-assembly. En: D. Stefanovic y A. Turberfield (eds.), *DNA Computing and Molecular Programming*, Vol. 7433 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 58–72.

Doty, D., Lutz, J., Patitz, M., Schweller, R., Summers, S., y Woods, D. (2012). The tile assembly model is intrinsically universal. En: *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, Oct. pp. 302–310.

Edgar, T. y Himmelblau, D. (1988). *Optimization of Chemical Processes*. McGraw-Hill. p. 672.

Eiben, A. y Smith, J. (2003). *Introduction to Evolutionary Computing*. Natural Computing Series. Springer.

Fu, T. J. y Seeman, N. C. (1993). DNA double-crossover molecules. *Biochemistry*, **32**(13): 3211–3220.

Fujibayashi, K., Hariadi, R., Park, S. H., Winfree, E., y Murata, S. (2008). Toward reliable algorithmic self-assembly of DNA tiles: A fixed-width cellular automaton pattern. *Nano Letters*, **8**(7): 1791–1797.

Gondro, C. y Kinghorn, B. P. (2007). A simple genetic algorithm for multiple sequence alignment. *Genetics and molecular research : GMR*, **6**(4): 964–982.

Göös, M. y Orponen, P. (2010). Synthesizing minimal tile sets for patterned dna self-assembly. En: Y. Sakakibara y Y. Mi (eds.), *DNA Computing and Molecular Programming*, Vol. 6518 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 71–82.

Head, T. (1987). Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bulletin of mathematical biology*, **49**(6): 737–59.

Hopcroft, J., Motwani, R., y Ullman, J. (2002). *Introducción a la teoría de autómatas, lenguajes y computación*. Pearson Educación.

Ignatova, Z., Martínez-Pérez, I., y Zimmermann, K.-H. (2008). *DNA Computing Models*. Springer Publishing Company, Incorporated.

Kari, L., Păun, G., Rozenberg, G., Salomaa, A., y Yu, S. (1998). Dna computing, sticker systems, and universality. *Acta Informatica*, **35**(5): 401–420.

Kari, L., Seki, S., y Sosik, P. (2012). *DNA Computing: Foundations and Implications*, Vol. 3 de *Handbook for Natural Computing*, pp. 1073–1127. Springer Verlag.

Karp, R. (1972). Reducibility among combinatorial problems. En: R. Miller y J. Thatcher (eds.), *Complexity of Computer Computations*. Plenum Press, pp. 85–103.

Lempiäinen, T., Czeizler, E., y Orponen, P. (2011). Synthesizing small and reliable tile sets for patterned dna self-assembly. En: L. Cardelli y W. Shih (eds.), *DNA Computing and Molecular Programming*, Vol. 6937 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 145–159.

Lewin, B. (2008). *Genes 9*. Número v. 9. Jones & Bartlett Learning. p. 892.

Lipton, R. (1995). DNA Solution of Hard Computational Problems. *Science*, **268**(5210): 542–545.

Ma, X. y Lombardi, F. (2008). Synthesis of tile sets for dna self-assembly. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **27**(5): 963–967.

McPherson, J. y Møller, S. (2000). *PCR*. Basics (BIOS Scientific Publishers). BIOS. p. 276.

Navarta, A. (2012). El misterio de las células nucleadas. Fecha de consulta: 15 de Abril de 2014. URL:http://www.clubcientificobezmiliana.org/blog/curiosidades-y-anecdotas/el-misterio-de-las-celulas-nucleadas/.

Nemhauser, G. y Wolsey, L. (1988). *Integer and combinatorial optimization*. Wiley-Interscience series in discrete mathematics and optimization. Wiley. p. 763.

Rothemund, P. W. K. y Winfree, E. (2000). The program-size complexity of self-assembled squares (extended abstract). En: *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, New York, NY, USA. ACM, STOC '00, pp. 459–468.

Rothemund, P. W. K., Papadakis, N., y Winfree, E. (2004). Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biology*, p. 2004.

Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N. V., Goodman, M. F., Rothemund, P. W. K., y Adleman, L. M. (1996). A sticker based model for DNA computation.

Sakamoto, K., Gouzu, H., Komiya, K., Kiga, D., Yokoyama, S., Yokomori, T., y Hagiya, M. (2000). Molecular Computation by DNA Hairpin Formation. *Science*, **288**(5469): 1223–1226.

Seckbach, J. (2012). *Genesis - In The Beginning: Precursors of Life, Chemical Models and Early Biological Evolution*. Cellular Origin, Life in Extreme Habitats and Astrobiology. Springer.

Smith, L., Corn, R., Condon, A., Lagally, M., Frutos, A., Liu, Q., y Thiel, A. (1998). A surface-based approach to DNA computation. *Journal of Computational Biology*, **5**(2): 255–267.

Wallet, B. C., Marchette, D. J., y Solka, J. L. (1996). Matrix representation for genetic algorithms. Vol. 2756, pp. 206–214.

Wang, H. (1960). Proving theorems by pattern recognition i. *Commun. ACM*, **3**(4): 220–234.

Watson, J. D. y Crick, F. H. (1953). Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, **171**: 737–738.

Winfree, E. y Bekbolatov, R. (2003). Proofreading tile sets: Error correction for algorithmic self-assembly. En: J. Chen y J. H. Reif (eds.), *DNA*. Springer, Vol. 2943 de *Lecture Notes in Computer Science*, pp. 126–144.

Winfree, E., Liu, F., Wenzler, L. A., y Seeman, N. A. (1998). Design and self-assembly of two-dimensional DNA crystals. *Nature*, **394**: 539–544.

Wolfram, S. (1983). Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, **55**: 601–644.

Yang, Y.-x., Wang, A.-m., y Ma, J.-l. (2009). A DNA Computing Algorithm of Addition Arithmetic. *2009 First International Workshop on Education Technology and Computer Science*, pp. 1056–1059.

Apéndice A. Conjuntos de mosaicos

En esta sección se presentan los conjuntos de mosaicos del mejor individuo obtenido para cada caso de prueba. Los conjuntos se presentan en formato de salida del AG-PATS que se describe a continuación:

- La primer línea indica el nombre del patrón seguido del tamaño de la instancia.
- En la segunda línea se presenta los parámetros utilizados en la experimentación: tamaño de población y número de generaciones.
- La tercer línea señala el tiempo transcurrido en segundos del procesamiento computacional que fue requerido en la experimentación del caso de prueba.
- En la cuarta línea se muestra el número de errores de fuerzas de pegado, el cual debe ser cero de acuerdo a las restricciones en el individuo.
- La línea cinco presenta el número de tipos de mosaicos en el individuo.
- Cada una de las siguientes líneas describe un tipo de mosaico del conjunto. El color se representa por 1 = negro y 0 = blanco. Las fuerzas de pegado para cada orientación del mosaico tienen la siguiente nomenclatura: ⟨ **Inicial de la orientación** ⟩ **:** ⟨ **Número de fuerza de pegado** ⟩. El `id` y `tipo` se usan para manejo interno del algoritmo, no influyen en el diseño del mosaico.

A.1. *Binary counter* de 6×6

```
   BinaryCounter6x6
int numPoblacion = 50;int generaciones = 150000;
(total time: 22 seconds)
error:0
tipos:4
color:0 id:1 N:5 S:4 W:2 E:2 tipo:1
color:1 id:2 N:4 S:4 W:3 E:3 tipo:2
```

```
color:0 id:3 N:5 S:5 W:3 E:3 tipo:3
color:1 id:4 N:4 S:5 W:3 E:2 tipo:4
```

A.2. *Binary counter* de 15×15

```
   BinaryCounter15x15
int numPoblacion = 50;int generaciones = 150000;
(total time: 105 seconds)
error:0
tipos:5
color:0 id:1 N:5 S:4 W:2 E:2 tipo:1
color:1 id:2 N:4 S:4 W:3 E:3 tipo:2
color:0 id:3 N:5 S:5 W:3 E:3 tipo:3
color:1 id:4 N:4 S:5 W:3 E:2 tipo:4
color:1 id:5 N:4 S:4 W:2 E:2 tipo:5
```

A.3. *Binary counter* de 32×32

```
   BinaryCounter32x32
int numPoblacion = 50;int generaciones = 150000;
(total time: 650 seconds)
error:0
tipos:19
color:0 id:1 N:5 S:4 W:2 E:2 tipo:1
color:1 id:2 N:4 S:4 W:3 E:3 tipo:2
color:0 id:3 N:5 S:5 W:3 E:3 tipo:3
color:1 id:4 N:4 S:5 W:3 E:2 tipo:4
color:0 id:5 N:8 S:7 W:1 E:1 tipo:5
color:1 id:6 N:7 S:7 W:6 E:6 tipo:6
color:0 id:7 N:8 S:8 W:6 E:6 tipo:7
color:1 id:8 N:7 S:8 W:6 E:1 tipo:8
```

```
color:0 id:9 N:12 S:11 W:9 E:9 tipo:9
color:1 id:10 N:11 S:11 W:10 E:10 tipo:10
color:0 id:11 N:12 S:12 W:10 E:10 tipo:11
color:1 id:12 N:11 S:12 W:10 E:9 tipo:12
color:0 id:13 N:16 S:15 W:13 E:13 tipo:13
color:1 id:14 N:4 S:15 W:14 E:14 tipo:14
color:0 id:15 N:16 S:16 W:14 E:14 tipo:15
color:1 id:16 N:15 S:16 W:14 E:13 tipo:16
color:0 id:17 N:20 S:19 W:17 E:17 tipo:17
color:1 id:18 N:19 S:19 W:18 E:18 tipo:18
color:0 id:19 N:20 S:20 W:18 E:18 tipo:19
```

A.4. *Chess board* de 6×6

```
   ChessBoard6x6
int numPoblacion = 50;int generaciones = 150000;
(total time: 30 seconds)
error:0
tipos:2
color:0 id:1 N:4 S:7 W:1 E:1 tipo:1
color:1 id:2 N:7 S:4 W:1 E:1 tipo:2
```

A.5. *Chess board* de 15×15

```
   ChessBoard15x15
int numPoblacion = 50;int generaciones = 150000;
(total time: 108 seconds)
error:0
tipos:11
color:0 id:1 N:4 S:7 W:1 E:1 tipo:1
color:1 id:2 N:7 S:4 W:1 E:1 tipo:2
```

```
color:1 id:3 N:2 S:3 W:1 E:2 tipo:3
color:0 id:4 N:3 S:2 W:1 E:2 tipo:4
color:0 id:5 N:6 S:5 W:2 E:1 tipo:5
color:1 id:2 N:5 S:6 W:2 E:1 tipo:6
color:1 id:1 N:8 S:7 W:1 E:5 tipo:7
color:0 id:2 N:7 S:8 W:1 E:5 tipo:8
color:0 id:1 N:9 S:2 W:5 E:1 tipo:9
color:1 id:2 N:2 S:9 W:5 E:1 tipo:10
color:0 id:2 N:4 S:4 W:1 E:1 tipo:11
```

A.6. Sierpinski de 6×6

```
   Sierpinski6x6
int numPoblacion = 50;int generaciones = 150000;
(total time: 22 seconds)
error:0
tipos:7
color:0 id:1 N:2 S:1 W:2 E:2 tipo:1
color:1 id:2 N:1 S:2 W:1 E:2 tipo:2
color:0 id:3 N:3 S:4 W:2 E:5 tipo:3
color:1 id:4 N:4 S:3 W:2 E:5 tipo:4
color:0 id:5 N:6 S:8 W:5 E:7 tipo:5
color:1 id:6 N:8 S:6 W:5 E:7 tipo:6
color:1 id:7 N:3 S:6 W:5 E:7 tipo:7
```

A.7. Sierpinski de 8×8

```
   Sierpinski8x8
int numPoblacion = 50;int generaciones = 150000;
(total time: 44 seconds)
error:0
```

```
tipos:9
color:0 id:1 N:2 S:2 W:2 E:1 tipo:1
color:1 id:2 N:1 S:1 W:1 E:1 tipo:2
color:0 id:3 N:2 S:1 W:2 E:2 tipo:3
color:1 id:4 N:1 S:2 W:1 E:2 tipo:4
color:0 id:5 N:3 S:4 W:2 E:5 tipo:5
color:1 id:6 N:4 S:3 W:2 E:5 tipo:6
color:0 id:7 N:6 S:8 W:5 E:7 tipo:7
color:1 id:8 N:8 S:6 W:5 E:7 tipo:8
color:1 id:9 N:3 S:6 W:5 E:7 tipo:9
```

A.8. Sierpinski de 12×12

```
   Sierpinski12x12
int numPoblacion = 50;int generaciones = 150000;
(total time: 101 seconds)
error:0
tipos:18
color:0 id:1 N:3 S:4 W:2 E:5 tipo:1
color:0 id:2 N:6 S:8 W:5 E:7 tipo:2
color:1 id:3 N:3 S:6 W:5 E:7 tipo:3
color:0 id:4 N:2 S:2 W:2 E:1 tipo:4
color:1 id:5 N:1 S:1 W:1 E:1 tipo:5
color:1 id:6 N:4 S:3 W:2 E:5 tipo:6
color:1 id:7 N:8 S:6 W:5 E:7 tipo:7
color:1 id:8 N:1 S:2 W:1 E:2 tipo:8
color:0 id:9 N:2 S:1 W:2 E:2 tipo:9
color:0 id:10 N:11 S:12 W:10 E:13 tipo:10
color:0 id:11 N:14 S:16 W:13 E:15 tipo:11
color:1 id:12 N:11 S:14 W:13 E:15 tipo:12
color:0 id:13 N:10 S:10 W:10 E:9 tipo:13
```

color:1 id:14 N:9 S:9 W:9 E:9 tipo:14

color:1 id:15 N:12 S:11 W:10 E:5 tipo:15

color:1 id:16 N:16 S:14 W:13 E:15 tipo:16

color:1 id:17 N:9 S:10 W:9 E:10 tipo:17

color:0 id:18 N:10 S:9 W:10 E:10 tipo:18

A.9. Sierpinski de 17×17

Sierpinski17x17

int numPoblacion = 50;int generaciones = 150000;

(total time: 288 seconds)

error:0

tipos:20

color:0 id:1 N:2 S:1 W:2 E:2 tipo:1

color:0 id:2 N:6 S:8 W:5 E:7 tipo:2

color:1 id:3 N:3 S:6 W:5 E:7 tipo:3

color:0 id:4 N:14 S:16 W:13 E:15 tipo:4

color:0 id:5 N:2 S:2 W:2 E:1 tipo:5

color:1 id:6 N:1 S:1 W:1 E:1 tipo:6

color:1 id:7 N:9 S:10 W:9 E:10 tipo:7

color:1 id:8 N:4 S:3 W:2 E:5 tipo:8

color:1 id:9 N:16 S:14 W:13 E:15 tipo:9

color:1 id:10 N:8 S:6 W:5 E:7 tipo:10

color:1 id:11 N:1 S:2 W:1 E:2 tipo:11

color:0 id:12 N:11 S:12 W:10 E:13 tipo:12

color:1 id:13 N:11 S:14 W:13 E:15 tipo:13

color:0 id:14 N:3 S:4 W:2 E:5 tipo:14

color:0 id:15 N:10 S:10 W:10 E:9 tipo:15

color:1 id:16 N:9 S:9 W:9 E:9 tipo:16

color:1 id:17 N:12 S:11 W:10 E:5 tipo:17

color:0 id:18 N:10 S:9 W:10 E:10 tipo:18

color:0 id:19 N:10 S:10 W:10 E:10 tipo:19

color:0 id:20 N:2 S:2 W:2 E:2 tipo:20

A.10. Sierpinski de 32×32

Sierpinski32x32

int numPoblacion = 50;int generaciones = 150000;

(total time: 1096 seconds)

error:0

tipos:23

color:0 id:1 N:2 S:2 W:2 E:2 tipo:1

color:0 id:2 N:2 S:1 W:2 E:2 tipo:2

color:0 id:3 N:14 S:16 W:13 E:15 tipo:3

color:1 id:4 N:9 S:9 W:9 E:9 tipo:4

color:0 id:5 N:2 S:2 W:2 E:1 tipo:5

color:1 id:6 N:9 S:10 W:9 E:10 tipo:6

color:1 id:7 N:1 S:1 W:1 E:1 tipo:7

color:1 id:8 N:4 S:3 W:2 E:5 tipo:8

color:0 id:9 N:10 S:10 W:10 E:10 tipo:9

color:0 id:10 N:6 S:8 W:5 E:7 tipo:10

color:1 id:11 N:3 S:6 W:5 E:7 tipo:11

color:1 id:12 N:16 S:14 W:13 E:15 tipo:12

color:1 id:13 N:1 S:2 W:1 E:2 tipo:13

color:0 id:14 N:11 S:12 W:10 E:13 tipo:14

color:1 id:15 N:11 S:14 W:13 E:15 tipo:15

color:0 id:16 N:3 S:4 W:2 E:5 tipo:16

color:1 id:17 N:8 S:6 W:5 E:7 tipo:17

color:0 id:18 N:10 S:10 W:10 E:9 tipo:18

color:1 id:19 N:12 S:11 W:10 E:5 tipo:19

color:0 id:20 N:10 S:9 W:10 E:10 tipo:20

color:1 id:21 N:12 S:11 W:1 E:2 tipo:21

color:0 id:22 N:11 S:12 W:2 E:1 tipo:22

color:1 id:23 N:14 S:14 W:14 E:11 tipo:23

A.11. Sierpinski de 64×64

Sierpinski64x64

int numPoblacion = 50;int generaciones = 150000;

(total time: 4872 seconds)

error:0

tipos:25

color:1 id:1 N:9 S:9 W:9 E:9 tipo:1

color:0 id:2 N:2 S:2 W:2 E:1 tipo:2

color:1 id:3 N:3 S:6 W:5 E:7 tipo:3

color:0 id:4 N:2 S:2 W:2 E:2 tipo:4

color:0 id:5 N:2 S:1 W:2 E:2 tipo:5

color:0 id:6 N:14 S:16 W:13 E:15 tipo:6

color:1 id:7 N:1 S:1 W:1 E:1 tipo:7

color:1 id:8 N:11 S:14 W:13 E:15 tipo:8

color:0 id:9 N:7 S:7 W:7 E:7 tipo:9

color:0 id:10 N:10 S:10 W:10 E:10 tipo:10

color:0 id:11 N:6 S:8 W:5 E:7 tipo:11

color:1 id:12 N:9 S:10 W:9 E:10 tipo:12

color:1 id:13 N:4 S:3 W:2 E:5 tipo:13

color:1 id:14 N:16 S:14 W:13 E:15 tipo:14

color:1 id:15 N:14 S:14 W:14 E:11 tipo:15

color:1 id:16 N:12 S:11 W:10 E:5 tipo:16

color:0 id:17 N:10 S:9 W:10 E:10 tipo:17

color:1 id:18 N:1 S:2 W:1 E:2 tipo:18

color:0 id:19 N:3 S:4 W:2 E:5 tipo:19

color:1 id:20 N:8 S:6 W:5 E:7 tipo:20

color:1 id:21 N:8 S:8 W:8 E:8 tipo:21

color:0 id:22 N:10 S:10 W:10 E:9 tipo:22

color:0 id:23 N:11 S:12 W:10 E:13 tipo:23

color:1 id:24 N:12 S:11 W:1 E:2 tipo:24

color:0 id:25 N:11 S:12 W:2 E:1 tipo:25

A.12. *Corner tree* de 23×23

CornerTree23x23

int numPoblacion = 50;int generaciones = 150000;

(total time: 401 seconds)

error:0

tipos:11

color:1 id:1 N:3 S:6 W:5 E:7 tipo:1

color:1 id:2 N:12 S:11 W:10 E:5 tipo:2

color:1 id:3 N:1 S:2 W:1 E:2 tipo:3

color:0 id:4 N:11 S:12 W:10 E:13 tipo:4

color:1 id:5 N:11 S:14 W:13 E:15 tipo:5

color:1 id:6 N:16 S:14 W:13 E:15 tipo:6

color:1 id:7 N:8 S:6 W:5 E:7 tipo:7

color:0 id:8 N:10 S:10 W:10 E:9 tipo:8

color:0 id:9 N:3 S:4 W:2 E:5 tipo:9

color:0 id:10 N:10 S:9 W:10 E:10 tipo:10

color:1 id:11 N:12 S:11 W:1 E:2 tipo:11

A.13. *Random* de 12×12

Random12x12

int numPoblacion = 50;int generaciones = 150000;

(total time: 67 seconds)

error:0

tipos:10

```
color:1 id:1 N:4 S:3 W:2 E:5 tipo:1
color:1 id:2 N:3 S:6 W:5 E:7 tipo:2
color:0 id:3 N:2 S:2 W:2 E:1 tipo:3
color:1 id:4 N:8 S:6 W:5 E:7 tipo:4
color:0 id:5 N:3 S:4 W:2 E:5 tipo:5
color:0 id:6 N:6 S:8 W:5 E:7 tipo:6
color:0 id:7 N:6 S:8 W:5 E:7 tipo:7
color:1 id:8 N:1 S:2 W:1 E:2 tipo:8
color:1 id:9 N:1 S:1 W:1 E:1 tipo:9
color:0 id:10 N:2 S:1 W:2 E:2 tipo:10
```

A.14. *Random* de 16×16

```
   Random16x16
int numPoblacion = 50;int generaciones = 150000;
(total time: 127 seconds)
error:0
tipos:17
color:0 id:1 N:4 S:7 W:1 E:1 tipo:1
color:1 id:2 N:8 S:7 W:1 E:5 tipo:2
color:0 id:3 N:7 S:8 W:1 E:5 tipo:3
color:0 id:4 N:3 S:2 W:1 E:2 tipo:4
color:1 id:5 N:10 S:11 W:9 E:10 tipo:5
color:0 id:6 N:11 S:10 W:9 E:10 tipo:6
color:0 id:7 N:9 S:2 W:5 E:1 tipo:7
color:1 id:8 N:2 S:9 W:5 E:1 tipo:8
color:0 id:9 N:4 S:4 W:1 E:1 tipo:9
color:0 id:10 N:12 S:15 W:9 E:9 tipo:10
color:1 id:11 N:15 S:12 W:9 E:9 tipo:11
color:1 id:12 N:7 S:4 W:1 E:1 tipo:12
color:1 id:13 N:2 S:3 W:1 E:2 tipo:13
```

color:0 id:14 N:14 S:13 W:10 E:9 tipo:14

color:1 id:15 N:13 S:14 W:10 E:9 tipo:15

color:1 id:16 N:8 S:15 W:9 E:13 tipo:16

color:0 id:17 N:6 S:5 W:2 E:1 tipo:17

A.15. *Random* de 20×20

Random20x20

int numPoblacion = 50;int generaciones = 150000;

(total time: 232 seconds)

error:0

tipos:21

color:0 id:9 N:16 S:14 W:13 E:15 tipo:1

color:0 id:10 N:8 S:6 W:5 E:7 tipo:2

color:0 id:11 N:1 S:2 W:1 E:2 tipo:3

color:0 id:12 N:11 S:12 W:10 E:13 tipo:4

color:1 id:13 N:11 S:14 W:13 E:15 tipo:5

color:0 id:1 N:2 S:1 W:2 E:2 tipo:6

color:0 id:2 N:6 S:8 W:5 E:7 tipo:7

color:1 id:3 N:3 S:6 W:5 E:7 tipo:8

color:0 id:4 N:14 S:16 W:13 E:15 tipo:9

color:0 id:5 N:2 S:2 W:2 E:1 tipo:10

color:1 id:6 N:1 S:1 W:1 E:1 tipo:11

color:0 id:7 N:9 S:10 W:9 E:10 tipo:12

color:1 id:8 N:4 S:3 W:2 E:5 tipo:13

color:1 id:16 N:9 S:9 W:9 E:9 tipo:14

color:1 id:17 N:12 S:11 W:10 E:5 tipo:15

color:0 id:18 N:10 S:9 W:10 E:10 tipo:16

color:0 id:19 N:10 S:10 W:10 E:10 tipo:17

color:0 id:20 N:2 S:2 W:2 E:2 tipo:18

color:0 id:14 N:3 S:4 W:2 E:5 tipo:19

color:0 id:15 N:10 S:10 W:10 E:9 tipo:20

color:0 id:2 N:6 S:8 W:5 E:7 tipo:21

A.16. *Random* de 32×32

Random32x32

int numPoblacion = 50;int generaciones = 150000;

(total time: 802 seconds)

error:0

tipos:31

color:0 id:1 N:2 S:2 W:2 E:2 tipo:1

color:0 id:2 N:2 S:1 W:2 E:2 tipo:2

color:0 id:3 N:10 S:9 W:10 E:10 tipo:3

color:0 id:4 N:9 S:2 W:5 E:1 tipo:4

color:1 id:5 N:11 S:14 W:13 E:15 tipo:5

color:0 id:6 N:7 S:7 W:7 E:7 tipo:6

color:0 id:7 N:10 S:10 W:10 E:10 tipo:7

color:0 id:8 N:6 S:5 W:2 E:1 tipo:8

color:1 id:9 N:5 S:6 W:2 E:1 tipo:9

color:0 id:10 N:6 S:8 W:5 E:7 tipo:10

color:1 id:11 N:9 S:9 W:9 E:9 tipo:11

color:0 id:12 N:2 S:2 W:2 E:1 tipo:12

color:1 id:13 N:3 S:6 W:5 E:7 tipo:13

color:1 id:14 N:9 S:10 W:9 E:10 tipo:14

color:1 id:15 N:4 S:3 W:2 E:5 tipo:15

color:1 id:16 N:16 S:14 W:13 E:15 tipo:16

color:1 id:17 N:14 S:14 W:14 E:11 tipo:17

color:1 id:18 N:12 S:11 W:10 E:5 tipo:18

color:1 id:19 N:1 S:2 W:1 E:2 tipo:19

color:0 id:20 N:10 S:10 W:10 E:9 tipo:20

color:0 id:21 N:11 S:12 W:10 E:13 tipo:21

color:1 id:22 N:12 S:11 W:1 E:2 tipo:22

color:0 id:23 N:7 S:8 W:1 E:5 tipo:23

color:0 id:24 N:11 S:12 W:2 E:1 tipo:24

color:0 id:25 N:4 S:7 W:1 E:1 tipo:25

color:1 id:26 N:7 S:4 W:1 E:1 tipo:26

color:1 id:27 N:2 S:3 W:1 E:2 tipo:27

color:0 id:28 N:3 S:2 W:1 E:2 tipo:28

color:0 id:29 N:14 S:16 W:13 E:15 tipo:29

color:1 id:30 N:1 S:1 W:1 E:1 tipo:30

color:1 id:31 N:8 S:7 W:1 E:5 tipo:31