Proyecto Fin de Grado
Ingeniería de Telecomunicación

Diseño de servidor para sistemas distribuidos sobre dispositivos Raspberry Pi

Autor: Álvaro Espejo Muñoz
Tutor: Sergio Luis Toral Marin

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Grado
Ingeniería de Telecomunicación

Diseño de servidor para sistemas distribuidos sobre dispositivos Raspberry Pi

Autor:
Álvaro Espejo Muñoz

Tutor:
Sergio Luis Toral Marin
Catedrático de Universidad

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Carrera: Diseño de servidor para sistemas distribuidos sobre dispositivos Raspberry Pi

Autor: Álvaro Espejo Muñoz

Tutor: Sergio Luis Toral Marin

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

Agradecimientos

Por supuesto agradecer a mi tutor, Sergio Luis Toral Marin, que me ha facilitado en todo lo posible la realización de este trabajo, pero también quiero dar unas palabras de agradecimiento a Daniel Gutierrez Reina, que a pesar del tiempo que ha pasado desde que solicité este trabajo siempre ha estado ayudándome y proporcionándome lecturas interesantes que sin duda me han sido útiles para poder completarlo.

Llegar hasta aquí no habría sido posible sin el cariño y la educación que mis padres, Roberto y María Dolores, me han sabido dar en su infinita paciencia conmigo.

Cómo no, agradezco a Merche por soportarme, sobre todo quererme, pero en especial ayudarme a terminar este grado en ingeniería, que sin su fuerza y ánimo no hubiera conseguido nunca.

A mi hermano Rober, por aconsejarme y escuchar siempre mis dudas y problemas.

Me acuerdo también de mis dos abuelos y dos abuelas que, allí donde estén, sé que sonreirán al verme convertirme en ingeniero.

A todos, gracias por el apoyo recibido durante toda esta larga etapa.

Resumen

El uso de algoritmos genéticos para resolver problemas de optimización combinatoria es una técnica cada vez más utilizada debido a que el espacio de búsqueda de soluciones es generalmente demasiado elevado como para ser resuelto por métodos convencionales. A menudo un algoritmo genético aplicado de forma secuencial se topa con la dificultad de que, al ser computacionalmente muy intensivo, en problemas donde se presenta una población de cierta envergadura se necesita de un tiempo de evaluación excesivamente elevado, lo que puede resultar poco adecuado para la resolución de problemas de optimización combinatoria. Una forma de extender la aplicabilidad de la computación evolutiva hacia problemas de mayor complejidad es la inclusión de paralelismo debido a las mejoras que presenta en cuanto a rendimiento.

Este trabajo está dividido en dos partes. La primera de ellas está dedicada a un estudio sobre las bases teóricas de dichos algoritmos, así como sus usos en computación en paralelo y computación distribuida en una red. En la segunda parte, se realiza un montaje de varios dispositivos Raspberry Pi conectados mediante Wi-Fi en una red de área local (LAN). Estos dispositivos serán configurados para trabajar de forma conjunta (distribuida) con el fin de resolver un problema de optimización clásico, el conocido como 'problema de las N reinas'. Usando esta estructura, se diseñará una serie de ensayos en el que se plantean diversos escenarios, haciendo uso de dichos algoritmos genéticos con el objetivo de paralelizar el algoritmo y poder realizar comparaciones entre los resultados obtenidos.

Como una de las conclusiones más reseñables de este trabajo podría decirse que existe un punto de inflexión a partir del cual es conveniente el uso de algoritmos genéticos paralelos con modelo de población basado en islas (para dispositivos con limitaciones computacionales, como puede ser una Raspberry Pi)

Abstract

The use of genetic algorithms to solve problems of combinatorial optimization is a technique that is being increasingly used because the search space is generally too big to be solved by conventional means. In many instances, genetic algorithms applied sequentially encounter the difficulty that an excessively high evaluation time (if the population is of a certain size) is required, which may be unsuitable for solving this kind of problems. One way to extend the applicability of evolutionary computing to more complex problems is to include parallelism, thanks to the performance improvements that it implies.

This work is divided in two parts. The first one is dedicated to a study on the theoretical bases of these algorithms, as well as their uses in parallel computing and distributed network computing. In the second part, several Raspberry Pi devices have been connected by Wi-Fi in a local area network (LAN). These devices will be configured to work together in order to solve a classic optimization problem, known as the 'N queens problem'. Using this structure, a series of tests will be designed (consisting of different scenarios that make use of these genetic algorithms) with the aim of parallelizing the algorithm and being able to make comparisons between the obtained results.

One of the most remarkable conclusions of this work could be that there is a turning point from which it is convenient to use parallel genetic algorithms with a population model based on islands (In the case of using devices with computational limitations, such as a Raspberry Pi).

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN Y OBJETIVOS

"We can only see a short distance ahead, but we can see plenty there that needs to be done".

- Computing Machinery and Intelligence (1950), Alan Turing -

En este primer capítulo se pretende dar contexto a la materia que se va a tratar durante este proyecto, los algoritmos genéticos, así como un pequeño resumen de su historia y actual estado del arte para, finalmente, introducir los principales objetivos que se pretenden alcanzar con el desarrollo del proyecto.

1.1 Introducción

Con el vertiginoso avance de las tecnologías en las últimas décadas, más concretamente desde la aparición de los primeros ordenadores de bajo coste a mediados de la década de 1980's, se ha impulsado el desarrollo de resolución a problemas que anteriormente la ingeniería era incapaz de abordar. A lo largo de este estudio vamos a centrarnos en una técnica que nació en dicho momento, se trata de los algoritmos evolutivos cuyo desarrollo desde entonces y hasta el día de hoy ha sido continuo.

Los algoritmos evolutivos se inspiran en los métodos de selección que usa la naturaleza, más concretamente en el principio darviniano de la selección natural. Actualmente han cobrado una gran importancia ya que por su evolución natural se han convertido en una poderosísima técnica de optimización y búsqueda, con una alta carga de computación en paralelo.

1.1.1 Trasfondo histórico. Selección natural y teoría de la evolución

En 1859 el científico naturalista inglés Charles Darwin publica "El origen de las especies por medio de la selección natural, o la preservación de las razas favorecidas en la lucha por la vida" dónde expone que las diversas especies biológicas compartimos una descendencia común que se ha ido ramificando a través de la evolución, sosteniendo que las especies evolucionan acorde al medio con el fin de adaptarse a éste. Esta rompedora visión diverge de la clásica visión de la naturaleza, estática y perfecta creada por Dios. En este escrito apareció por primera vez el concepto Selección Natural, definido como aquel medio que permite explicar la evolución biológica de la siguiente forma (Darwin, 1859): Existen organismos que se reproducen y la progenie hereda características de sus progenitores, existen variaciones de características si el medio ambiente no admite a todos los miembros de una población en crecimiento. Entonces aquellos miembros de la población con características menos adaptadas (según lo determine su medio ambiente) morirán con mayor probabilidad. Entonces aquellos miembros con características mejor adaptadas sobrevivirán más probablemente.

La teoría de la evolución de Charles Darwin se sustenta en tres premisas: La primera es que debe existir *variabilidad* del rasgo entre los individuos de una población, es decir ha de presentar diferentes estados o

fenotipos; La segunda, el rasgo debe ser sujeto a *transmisión*, es decir cada fenotipo de dicha característica ha de poder ser transmitido; La tercera y última, la variabilidad del rasgo debe dar lugar a *diferencias* en la supervivencia, es decir debe de haber algún factor que afecte a la probabilidad de que algunas características de nueva aparición se pueda extender en la población.

Si se cumplen estas tres premisas, puede tener lugar el fenómeno de la selección natural. En caso de no cumplirse cualquiera de ellas, no puede haber selección natural.

August Friedrich Leopold Weismann, científico biólogo alemán formuló (Weismann, 1892) en el siglo XIX la teoría de plasma germinal, o germoplasma, en la cual distinguía entre células germinales, que eran capaces de transmitir información hereditaria, y células somáticas, que no transmitían información alguna.

Gregor Johann Mendel, científico, monje agustino católico y Abad de la abadía de Santo Tomás de Brno realizó una serie de experimentos (Mendel, 1865) con guisantes durante un largo periodo de su vida, los cuales estudió y le sirvió para enunciar a partir de ellos las leyes básicas que gobiernan la herencia y que son conocidas como leyes de mendel.

Hoy en día se conoce como paradigma Neo-Darwiniano a la conjución de la teoría evolutiva original de Charles Darwin junto al seleccionismo de August Weismann y la genética de Gregor Mendel.

El Neo-Darwinismo establece (Hoffmann, 1989) que con sólo 4 procesos se puede explicar la mayoría de vida en nuestro planeta.

- Reproducción
- Mutación
- Competencia
- Selección

1.1.2 Trasfondo histórico. Computación Evolutiva

Los primeros hechos relacionados con algoritmos genéticos surgen en 1942, cuando Walter Bradford Cannon interpreta la evolución natural como un aprendizaje similar al ensayo y error (Cannon, 1954).

También en 1948, el matemático, lógico, científico de la computación, criptógrafo y filósofo Alan Turing establece una conexión entre evolución y aprendizaje máquina, en un informe supervisado por el mismo Sir Charles Darwin dónde proponía máquinas llamadas unorganized machines o u-machines (Turing, 1948).

No obstante la apertura en el desarrollo de los algoritmos genéticos se da, durante principios de los años sesenta, cuando distintos investigadores desarrollan de manera independiente algoritmos inspirados en la evolución. El profesor Ingo Rechenben, de la Universidad Técnica de Berlin, introdujo la que llamó Estrategia Evolutiva. Una técnica sin población ni cruce; un padre mutaba para producir un descendiente y se conservaba el mejor de los dos, convirtiéndose en padre de la siguiente iteración. En 1966 Lawrence J. Fogel desarrolla la Programación Evolutiva, donde se introduce por primera vez el concepto de población, que permitía que la salida de resultados no dependiera sólo de la entrada de datos actuales sino también de las anteriores (Fogel, Owens, & Walsh, 1966).

Sin embargo, en estas dos metodologías se obvia un fenómeno fundamental de la propia evolución, el cruce. Fue el profesor de Filosofía, Ingeniería Eléctrica y Ciencias de computación, John H. Holland de la Universidad de Míchigan el primero en plantear la posibilidad de adaptar e incorporar mecanismos naturales de selección y supervivencia a la resolución de problemas computacionales, presentando por primera vez de forma rigurosa y sistemática el concepto de sistemas digitales adaptativos con mecanismos de mutación, selección y cruce. Holland es considerado el 'Padre de los algoritmos genéticos' por desarrollar (Holland, 1975) dos puntos clave:

- Imitar los procesos adaptativos de la naturaleza.
- Diseñar un sistema informático capaz de usar mecanismos y técnicas de los sistemas naturales para resolver problemas

Esta investigación fue fundamentalmente teórica, siendo su realización práctica muy difícil y limitada en aquella década. El trabajo de John Holland da como resultado una técnica de optimización estocástica

originalmente llamada “planes reproductivos” y tachada como técnica no convencional de optimización para problemas del mundo real. No obstante, a partir de la creación de estas estrategias evolutivas aparecieron otras vías de investigación, siendo la más extendida la conocida como algoritmos genéticos, llevada a cabo por David Goldberg, Ingeniero Industrial de la Universidad de Illionis y alumno de John Holland.

Goldberg definiría al algoritmo genético como (Goldberg, Genetic algorithms in search, optimization, and machine learning, 1989): “Algoritmos de búsqueda basados en la mecánica de selección natural y de la genética natural. Combinan la supervivencia del más apto entre estructuras de secuencias con un intercambio de información estructurado, aunque aleatorizado, para constituir así un algoritmo de búsqueda que tenga algo de las genialidades de las búsquedas humanas”

1.1.3 Estado actual

A menudo un algoritmo genético aplicado de forma secuencial se topa con la dificultad de que, al ser computacionalmente muy intensivos, en problemas donde se presenta una población de cierta envergadura se necesita de un tiempo de evaluación excesivamente elevado que en ciertos casos imposibilita su implementación. Un ejemplo de ello lo podemos ver en el trabajo (Burgener & Storti, 2004) en el que plantea un problema de optimización para un sistema de refrigeración en el que con una población de 500 individuos, se descarta el uso del algoritmo secuencial ya que se requiere un tiempo de procesamiento por generación extremadamente elevado, aproximadamente 623 minutos.

Una forma de extender la aplicabilidad de la computación evolutiva hacia problemas de mayor complejidad es la inclusión de paralelismo, debido a las mejoras que presenta en cuanto a rendimiento. Aplicado al caso anterior, para la misma situación planteada, haciendo uso de un algoritmo en paralelo se requiere un tiempo aproximado de 83 minutos.

Esto se debe a dos factores, el primero, una ejecución en paralelo permite un mayor uso de recursos de cómputo como tiempo de procesador o memoria. El segundo factor es que el mismo comportamiento del algoritmo cambia en consecuencia de la estructura de la población, la ejecución en paralelo del algoritmo evolutivo permite una estructura poblacional formada por varios grupos interconectados de individuos. El balance final que se pretende encontrar es que se obtenga no solo un tiempo de espera menor para obtener la solución del problema si no una mejor calidad de búsqueda y del resultado obtenido.

Sin embargo, en el paso de un algoritmo genético secuencial a un algoritmo genético en paralelo no todos los elementos son susceptibles de paralelizarse. Es claro que la evaluación de la bondad de los individuos es una tarea cuya paralelización no afecta al comportamiento del algoritmo. No obstante, el esquema de un algoritmo genético es muy secuencial ya que el operador selección sí debe ser aplicado de foma global a toda la población lo que dificulta la total paralelización del problema.

El uso de los algoritmos evolutivos hoy en día está presente en muchos y diversos campos, como lo son medicina, desarrollo informático o incluso otros más alejados de la técnica como ciencias sociales o análisis lingüistico, dada su gran utilidad en problemas de tipo multivariable. Ejemplos de sus aplicaciones dentro del campo de la Medicina puede darse en sistemas con sistema de soporte de decisión, p.e medicina oftalmológica (Krawiec & Pawlak, 2015) y medicina oncológica (Conor, Fitzgerald, Medernach, & Krawiec, 2015). Dentro del campo de desarrollo informático, un ejemplo podría ser en uso de sistemas de búsqueda bugs (Yang, Jeong, Min, Lee, & Lee, 2018). En campos más alejados de la técnica, podemos ver ejemplo en sociología, dónde se pueden modelar evoluciones de pensamiento social (Khan, Streater, Bathia, Fiore, & Bölöni, 2013) o economía, usándolos para modelar el comportamiento de agentes del mercado (Markowska-Kaczmar, Kwasnicka, & Szczepkowski, 2008).

A continuación se lista las principales áreas en las que más está creciendo el uso de los algoritmos evolutivos.

- Redes Neuronales: De la fusión entre redes neuronales y algoritmos evolutivos obtenemos neuroevolución, un conjunto de técnicas y métodos que evolucionan redes neuronales artificiales.
- Evolve Hardware: Se trata de un campo centrado en el uso de algoritmos evolutivos para crear electrónica especializada sin necesidad de ingeniería manual. Engloba campos como hardware reconfigurable, computación evolutiva, tolerancia a fallos y sistemas autónomos.
    - Ejemplo clásico de Ingeniería Aeroespacial (Petit, 1998): Andrew Keane, profesor de ingeniería en la Universidad de Southampton en Inglaterra, utilizó los algoritmos genéticos

para producir un nuevo diseño para una viga de carga que pudiese montarse en órbita y utilizarse en satélites y estaciones espaciales. Tras quince generaciones el resultado de su estudio fueron 4.500 diseños diferentes de estructuras retorcidas que ningún ingeniero humano diseñaría. Entre ellas el mejor modelo tenía un aspecto irregular y, en cierta forma, orgánico que Keane comparó con un fémur humano. Este diseño resultó ligero, fuerte y con gran capacidad de amortiguación de vibraciones perjudiciales. Las pruebas en modelos confirmaron su superioridad a los diseñados por humanos como un soporte estable. Sin embargo "Ninguna inteligencia produjo los diseños. Simplemente evolucionaron"

- Ejemplo clásico de diseño de antena (Lohn, Hornby, & Linden, 2005): Los investigadores han investigando el diseño y optimización de antenas evolutivas desde principios de la década de 1990s. Hoy en día la velocidad de computación ha aumentado y los simuladores electromagnéticos han mejorado en los últimos años. Muchos tipos de antenas han sido investigados haciendo uso de algoritmos genéticos, incluyendo antenas de alambre con propiedades especificadas a priori, investigadas por Altshuler y Linden en 1997, Arrays de antenas, investigadas por Haupt en 1996, ó antenas cuadrifilares helicoidales (QFH), estudiadas por Jason Lohn en 2002.

Por otra parte, en los tiempos recientes millones de dispositivos con implementación de IOT (Internet de las Cosas) que nos rodea presentan un problema: son dispositivos de media o baja capacidad de computación que recolectan información, pero no hacen nada con ella. La envían, por ejemplo, a la nube, donde grandes centros de datos (súper computadoras) la procesan para obtener ciertas conclusiones o activar ciertas acciones. Es por esto que se está imponiendo una nuevo paradigma de computación distribuida, denominado Edge Computing, en la que dicha computación se realiza en parte (o completamente) en nodos de dispositivos distribuidos a los que se les denomina dispositivos inteligentes o dispositivos de borde (Edge) en lugar de tener lugar principalmente en un entorno de nube centralizada.

La principal motivación es proporcionar recursos de servidor, como pueden ser análisis de datos e inteligencia artificial, más cercanos a las fuentes de recopilación de datos y sistemas ciberfísicos, como por ejemplo sensores inteligentes. El Edge Computing se considera una herramienta importante en la realización de campos tales como computación física, contribución a las ciudades inteligentes, computación ubicua, aplicaciones multimedia como VR (realidad aumentada) y juegos en la nube, así como el Internet de las cosas.

1.2 Objetivos

El objetivo principal de este proyecto es realizar un análisis del funcionamiento de los algoritmos genéticos en un escenario distribuido. Para esta tarea en primer lugar se va a realizar un estudio previo sobre las bases teóricas de dichos algoritmos, así como sus usos en computación en paralelo y computación distribuida en una red.

En segundo lugar se va a realizar un montaje de varios dispositivos Raspberry Pi conectados mediante Wi-Fi en una red de área local (LAN). Estos dispositivos serán configurados para trabajar de forma conjunta (distribuida) con el fin de resolver un problema de optimización clásico, el conocido como ‘problema de las N reinas’, haciendo uso de dichos algoritmos genéticos con el objetivo de paralelizar el algoritmo y poder realizar comparaciones entre los resultados obtenidos. La elección de solucionar este problema haciendo uso de algoritmos genéticos no es casual, pues entre las diferentes técnicas de computación evolutiva la de los algoritmos genéticos con representación ordinal (Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning., 1989) es la utilizada en la resolución de problemas de optimización combinatoria, donde el espacio de búsqueda se compone de permutaciones (ordenamiento de números enteros).

Seguidamente se realizará un estudio del software disponible para implementar técnicas relacionadas con algoritmos genéticos y su posterior aplicación para la resolución del problema ya mencionado de las N reinas.

Finalmente se diseñará una serie de ensayos haciendo uso de todos lo anterior descrito en el que se plantean diversos escenarios, de los cuales se va a realizar un análisis comparativo de resultados.

2 FUNDAMENTOS TEÓRICOS

Una vez se ha explicado el trasfondo a cerca de los algoritmos genéticos, en qué se inspiran, su origen y porqué funcionan bien, toca explicar de forma concisa los procedimientos con los que se sirve.

En primer lugar se va a comentar, de forma resumida, la base biológica en la que se sustentan los algoritmos genéticos para. Seguidamente, entraremos de lleno en el concepto y componentes fundamentales que conforman, desde un punto de vista computacional, un algoritmo genético: Inicialización, selección, cruce, reemplazo, elitismo, mutación y métodos de evaluación.

A continuación, comentaremos diferentes estrategias de implementación, haciendo hincapié en aquellas que utilizan procesamiento en paralelo. Por último en este capítulo se va desarrollar en profundidad el problema tipo que vamos a utilizar durante el desarrollo de este trabajo, el llamado 'Problema de las N Reinas'.

2.1 Base Biológica

Tal y como se ha comentado anteriormente, los algoritmos genéticos se basan en la evolución biológica y su base genético-molecular. La evolución es un proceso que opera sobre cromosomas. Un cromosoma se puede considerar como la herramienta orgánica que codifica la vida y aquella información contenida en los cromosomas se conoce como genotipo. En genética se define a fenotipo como aquella expresión del genotipo en función de un determinado ambiente (Oliva, Ballesta, Oriola, & Clària, 2008), en otras palabras, son aquellas diferencias físicas y de comportamiento que afectan a la respuesta de un individuo a su entorno.

En la naturaleza, los distintos rasgos fenotípicos vienen determinados tanto por herencia como por factores relacionados con el desarrollo del individuo. Es decir, si los rasgos fenotípicos incrementan la posibilidad de reproducción y además son heredables, dichos rasgos tienden a incrementar en las subsecuentes generaciones, sirviendo de base a nuevas combinaciones de rasgos.

Durante la fase de reproducción ocurre el llamado proceso evolutivo. Los mecanismos más importantes a estudiar en esta introducción son: Mutación, en la cual los cromosomas de padres e hijos resultan diferentes, y Cruce (o recombinación), que combinan los cromosomas de los padres para producir la descendencia. La combinación de buenas características de heredadas puede originar que el individuo esté mejor adaptado al medio que sus ancestros.

2.2 Algoritmos genéticos

La idea básica de un algoritmo genético nace en aquellos problemas donde no podemos usar un método resolutivo clásico.

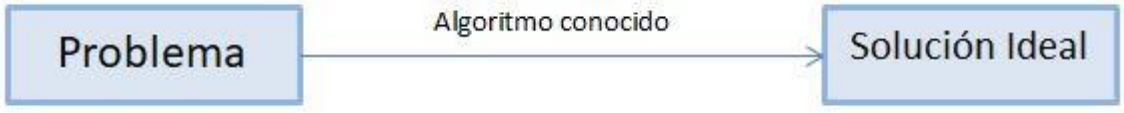

Figura 2-1. Esquema de resolución de un problema clásico.

En el método resolutivo clásico hemos de conocer el algoritmo a usar para resolver el problema mientras que la selección natural elimina uno de los mayores obstáculos en el diseño software: especificar de antemano todas las características de un problema y las acciones que el programa habría de tomar para enfrentarlo. Al

aprovechar los mecanismos de la evolución se pueden resolver problemas cuya estructura nos es desconocida.

Un algoritmo evolutivo es capaz de resolver problemas de tipo NP, siglas en inglés para *nondeterministic polynomial time,* es decir problemas que no pueden resolverse en un tiempo polinómico. Este tipo de problema engloba problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las ya conocidas, es decir, se debe realizar una búsqueda exhaustiva de todas las posibilidades del espectro.

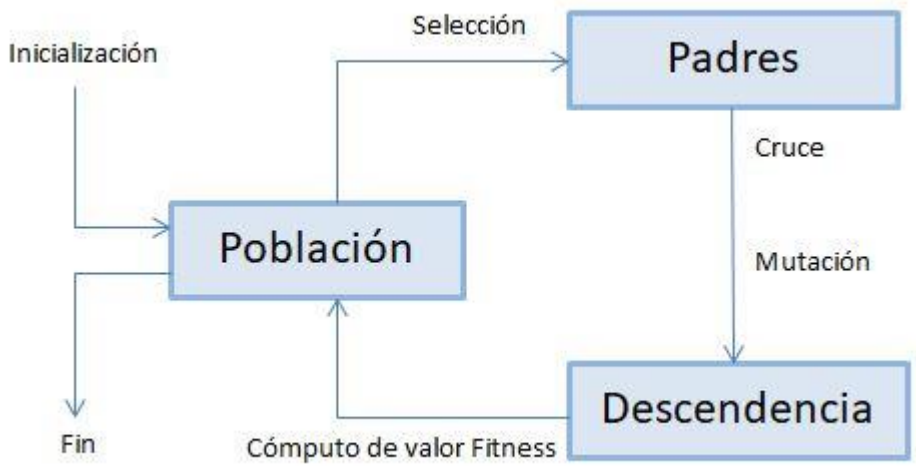

Figura 2-2. Esquema de resolución de un problema mediante el uso de Algoritmo Evolutivo Simple.

Durante el proceso evolutivo se trabaja sobre una población de individuos, donde cada uno representa una posible solución al problema que se desea resolver. A cada individuo se le asocia un valor de ajuste, denominado fitness, que cuantifica la validez de la solución respecto al problema. Buscando una analogía con la propia naturaleza, podría decirse que el valor de ajuste sería el equivalente a cuantificar la eficiencia de un individuo en aprovechar los recursos del medio.

2.2.1 Inicialización

Se trata de la primera parte del algoritmo y es el encargado de elegir de entre todo el espectro posible de soluciones que el problema puede aceptar. La elección se hace de una forma aleatoria tal que se consiga una población inicial lo más heterogénea que se pueda, representando la diversidad que presenta el sistema y llegando así a los óptimos de manera más rápida y abarcando todo el dominio.

Para evaluar los individuos de forma efectiva y atendiendo a las necesidades específicas de cada problema se usa un tipo de codificación específica. La codificación es pues una representación, elegida, del formato de la solución.

2.2.2 Selección

Parte del algoritmo que se encarga de escoger cuál individuo va a tener oportunidad de reproducirse y cuál no. Para ello se ha de tener en cuenta que se ha de otorgar una mayor tasa de probabilidad de reproducción a aquellos individuos más aptos sin descartar a los menos aptos, ya que esto podría provocar que la población se volviese homogénea en pocas generaciones.

Existen dos grupos en los que se puede dividir los algoritmos de selección: Probabilísticos y Determinísticos.

2.2.2.1 Algoritmos de selección probabilísticos

En este tipo de algoritmos se adjudica la probabilidad de selección con una componente de aleatoriedad. Es en este grupo donde se encuentran los algoritmos de selección por ruleta o por torneo que, dada su importancia por ser los más utilizados, se describen a continuación.

- **Selección por ruleta** (Blickle & Thiele, 1995)**:** A cada individuo le es asignada una parte proporcional a su valor de ajuste en una "ruleta" de tal forma que la suma de los porcentajes asignados a cada individuo sea la unidad y que los mejores individuos recibirán una porción de la ruleta mayor que la recibida por los peores. Generalmente la población está ordenada en base al ajuste, por lo que las porciones más grandes se encuentran al inicio de la ruleta. Para seleccionar a un individuo se genera un número aleatorio comprendido [0,1] y se escoge el individuo situado en dicha posición de la ruleta. A pesar de ser, probablemente, el método de selección más conocido por ser muy sencillo es altamente ineficiente a medida que aumenta el tamaño de la población.
- **Selección por torneo probabilístico** (Miller & Goldberg, 1995): Este método consiste en hacer comparaciones entre individuos. Primero se selecciona un número p de individuos (generalmente se escoge p=2 para hacer comparaciones dos a dos) de manera aleatoria y luego se genera un número al azar en el intervalo [0,1]. Si dicho valor es mayor a un parámetro q fijado para todo el proceso evolutivo, que suele comprender el rango $0.5 < q \leq 1$, se escoge el individuo más apto o en caso contrario se escoge el menos apto.

2.2.2.2 Algoritmos de selección determinísticos

Dependiendo del número de veces que se tomen los mejores y peores individuos las distintas variaciones en este tipo de algoritmos permiten hacer una búsqueda más intensiva en cierto lugar del espectro (por ejemplo si se seleccionan muchas más veces aquellos con mejor valor de ajuste) o que se reparta la búsqueda en todo el dominio.

- Selección por torneo determinístico: Al igual que en el caso probabilístico este algoritmo se basa en hacer comparaciones entre individuos, seleccionando a un número p de individuos (generalmente se escoge p=2 para hacer comparaciones dos a dos) de manera aleatoria. La diferencia radica en la forma de elegir el ganador, aquí directamente gana aquel cuyo valor de ajuste sea más alto.

2.2.3 Cruce

Tal y como ocurre en la naturaleza una generación proviene de otra anterior, así una vez seleccionados los individuos éstos han de ser recombinados para producir la descendencia que se insertará en la siguiente generación. El operador de cruce equivale a una reproducción de tipo sexual en la naturaleza. Se trabajan con probabilidades, siendo éstas de gran importancia pues independientemente del algoritmo un alto porcentaje de cruce hace que el algoritmo tienda a perder rápidamente heterogeneidad y, por tanto, es conveniente dejar parte de la población sin cruce para conservar diversidad y evitar converger hacia óptimos locales.

- Cruce basado en un punto (comúnmente conocido como SPX, Single Point Exchange): Los dos individuos seleccionados como progenitores son recombinados por medio de la selección de un único punto de corte seleccionado aleatoriamente. El resultado será la misma información a la izquierda pero intercambiarán el contenido que se encuentran a la derecha.
- Cruce de doble punto (comúnmente conocido como DPX, Double Point Crossover): Los dos individuos seleccionados como progenitores son recombinados por medio de dos cortes no coincidentes, garantizando que se originen tres segmentos. Por normal general para generar los individuos resultantes se escoge el segmento central de uno de los individuos y los segmentos de cola del otro.

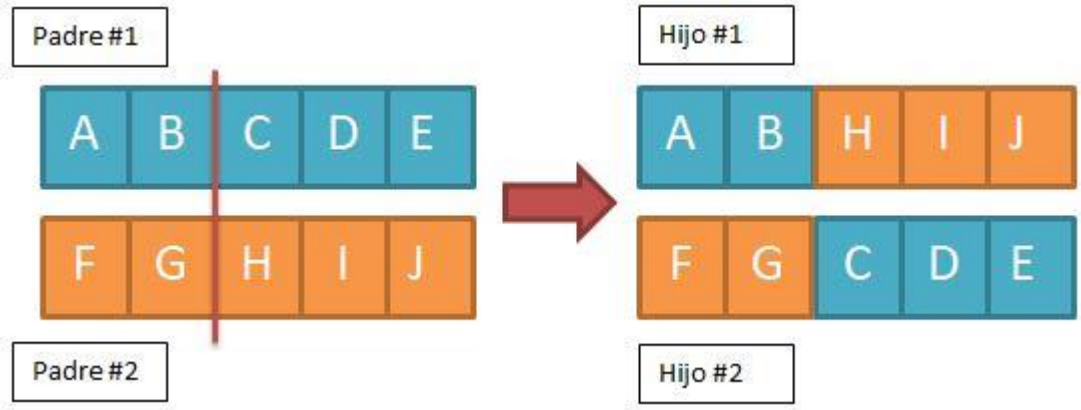

Figura 2-3. Esquema simple representando un cruce de un único punto.

Aunque se pueden seguir añadiendo más de dos puntos de cruce, denominadas técnicas de cruce multipunto, Kenneth A. De Jong investigó (De Jong, 1975) el comportamiento del operador de cruce basado en múltiples puntos y llegó a la conclusión que el doble punto de cruce supone una mejoría respecto a un único punto de cruce pero, no obstante, añadir más puntos de cruce no mejoraba el comportamiento del algoritmo. Es más, colocar demasiados puntos de cruce añadía el problema de que los segmentos resultantes sean corrompibles, esto es, que puede que pierdan las características de "bondad" que poseían los segmentos en su conjunto. Aun así, no todo son desventajas ya que si se añaden más puntos de cruce se consigue que el espacio de búsqueda del problema sea explorado de forma más intensidad

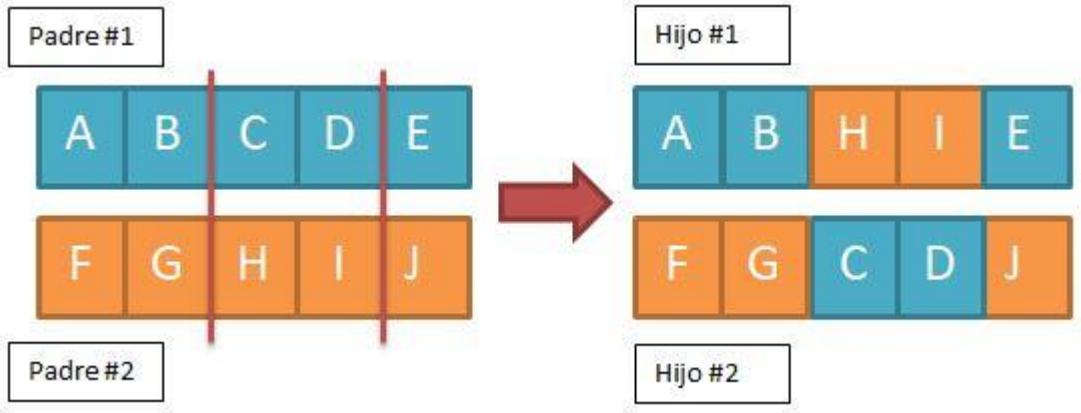

Figura 2-4. Esquema simple representando un cruce de doble punto

Alternativamente se pueden hacer uso de otros mecanismos de reproducción distintos al cruce como puede ser la Copia, que equivale a un mecanismo de reproducción asexual presente en la naturaleza. Este mecanismo consiste en que de forma azarosa un individuo pueda tener una copia íntegra en la nueva generación. La probabilidad de que esto ocurra ha de ser menor a la probabilidad de cruce pues numerosas copias dentro de una generación puede conllevar a una convergencia prematura.

2.2.4 Reemplazo

Existen varias maneras de actuar una vez sean recombinados los individuos según su candidatura para pasar a la siguiente generación. Decimos que estas dos maneras de actuar se denominan técnicas destructivas o no destructivas.

- En una estrategia destructiva los descendientes se insertan en la siguiente población generacional independientemente del valor de ajuste que tuvieran sus padres.
- En una estrategia no destructiva la descendencia solamente pasa a la siguiente generación si su valor de ajuste mejora a la de sus padres.

2.2.5 Elitismo

El elitismo es una técnica que asegura que en cada generación haya una mejora, nunca dando un paso atrás en cuanto a la calidad de la mejor solución. Para conseguir su propósito ésta técnica consiste en que en el paso de una generación a la siguiente se copie tal cual al mejor individuo (o mejores).

2.2.6 Mutación

La mutación es un mecanismo basado en cambiar el valor de ciertos genes, por lo general únicamente uno, por otros de forma aleatoria. Esta aleatoriedad es configurable y permite, si la probabilidad es alta, disponer de una mayor heterogeneidad en la población pero mayor complejidad en el cómputo.

Aunque se pueda realizar mutación en un individuo antes de introducirlos en la nueva población en algoritmos genéticos es habitual hacerlo junto con el operador cruce.

Este método, que presenta como objetivo ofrecer diversificación en la población, es brusco y poco habitual en la naturaleza por lo que su uso en algoritmos genéticos es tal que, generalmente, su probabilidad de mutación es muy baja o por lo menos muy inferior a la de cruce. Esto se debe a que, por lo general, tras la mutación el valor de ajuste suele ser menor. No obstante las mutaciones permiten que ningún punto del espacio tenga nula posibilidad de ser explorado.

2.2.6.1 Mutación conjunta con cruce

Como se ha mencionado previamente es habitual hacer uso conjunto de Cruce + Mutación. Esta sencilla operación conjunta se da de la siguiente forma: Se seleccionan, con probabilidad X, dos individuos de la población como padres para realizar la operación de cruce. Solamente si el cruce tiene éxito entonces uno o ambos hijos se someten a la operación de mutación, con cierta probabilidad Y.

2.2.6.2 Mecanismos de mutación

La mutación más usual es el reemplazo aleatorio, esto es, variar con cierta probabilidad un gen de un cromosoma. Que en algoritmos genéticos equivaldría a:

En codificaciones binarias

- Negar el bit que ha sido aleatoriamente seleccionado.

En otras codificaciones no binarias:

- Simplemente intercambiar los valores de dos genes de un mismo individuo (Ya hablaremos más tarde de este método en el epígrafe 2.4.4)
- Incrementar o reducir el valor de un gen en una pequeña cantidad aleatoria.
- Multiplicar un gen por un valor decimal en torno a 1.

2.2.7 Evaluación (Cómputo de valor de fitness)

Hemos de saber si los individuos de la población son o no buenas soluciones candidatas al problema, para ello necesitamos un método de evaluación. No existe un método de evaluación que sirva para todos los tipos de problemas, por ello cada uno ha de tener un método de evaluación. Además ocurre lo mismo dependiendo de cómo se codifiquen las soluciones, es decir, dependiendo de cómo sean los individuos deberá derivarse un método de evaluación distinto.

Los algoritmos de evaluación dan un dato numérico a la bondad de cada solución que recibe el nombre de valor de ajuste o fitness y se empleará para llevar un control de los individuos tras la aplicación de operadores genéticos.

John R. Koza, científico informático, ex profesor adjunto en la Universidad de Stanford y conocido por su notable labor como pionero en el uso de la programación genética como herramienta para optimización de problemas complejos, diferencia cuatro tipos de valores de ajuste o fitness (Koza, 1994), ellos son fitness puro, fitness estandarizado, fitness ajustado y fitness normalizado.

2.2.7.1 Raw Fitness (Fitness puro)

$$r(i,t) = \sum_{j=1}^{N_c} |s(i,j) - c(i,j)| \qquad (2-1)$$

Correspondiendo la expresión anterior al cálculo del valor de fitness puro dónde S(i,j) es el valor que devuelve una expresión S de un individuo i en una generación t y C(j) es un valor de fitness que, conocemos, es correcto.

En problemas de maximización aquellos individuos con valor de fitness puro alto serán los más interesantes, mientras que en un problema de minimización interesan los individuos con valor de función objetivo reducido.

2.2.7.2 Standardized Fitness (Fitness estandarizado)

Para solucionar la dualidad en problemas de minimización o maximización se modifica la función objetivo según interese. Si, para un problema en particular, interesa el menor valor de fitness puro, el valor de fitness estandarizado es igual al valor de fitness puro para ese problema. En cambio si, para un problema particular, interesa el mayor valor de fitness puro, hay que calcular el valor de fitness estandarizado a partir del valor de fitness puro. Para ello el valor de fitness puro se resta de una cota superior r_{max} y, así, la bondad de un individuo será mayor cuando más cercano esté a cero el valor del ajuste.

$$s(i,t) = \begin{cases} r(i,t) & minimización \\ r_{max} - r(i,t) & maximización \end{cases} \qquad (2-2)$$

Dónde la expresión anterior se corresponde con el cálculo del valor de fitness estandarizado de un individuo i en una generación t y r(i,t) es el valor de fitness puro.

2.2.7.3 Adjusted Fitness (Fitness ajustado)

El fitness ajustado toma siempre valores del intervalo (0,1]. Cuanto más se aproxime el fitness ajustado a 1 mayor es su bondad. El fitness ajustado tiene la ventaja de exagerar la importancia de las pequeñas diferencias en el fitness estandarizado cuando dicho valor fitness estándar se aproxima a 0, lo cual tiende a ocurrir en las últimas generaciones del proceso. Así, a medida que la población mejora mayor hincapié se hace en las pequeñas diferencias que marcan la diferencia entre un buen individuo y un muy buen individuo.

$$a(i,t) = \frac{1}{1 + s(i,t)} \qquad (2-3)$$

2.2.7.4 Normalized Fitness (Fitness normalizado)

Si el método de selección empleado es proporcional al valor de fitness es necesario el concepto de fitness normalizado. El fitness normalizado introduce indica la bondad de una solución respecto a las demás de la población.

$$r(i,t) = \frac{a(i,t)}{\sum_{k=1}^{N_c} a(k,t)} \qquad (2-4)$$

Sus características son las siguientes

- Varía (0,1], al igual que el fitness ajustado, cuyos mejores individuos serán los más próximos a la unidad
- Un valor cercano a uno no sólo significa que sea una buena solución al problema sino que además es mucha mejor solución que el resto de población.
- La suma de los valores de fitness normalizado es 1.

2.3 Estrategias de implementación de algoritmos genéticos

Los Algoritmos Genéticos tienen la capacidad de ser eficaces ante un amplio abanico de problemas que abarcan diferentes áreas. Aunque no se garantiza que un Algoritmo Genético encuentre la solución óptima del problema es capaz de encontrar soluciones de nivel aceptable en un tiempo adecuado. En un determinado tipo de problema es posible que exista una técnica especializada para resolverlo que mejore en rapidez o eficacia a un algoritmo genético, no obstante el mayor punto a favor de un Algoritmo Genético es su aplicación en aquellos tipos de problemas por los cuales no existe ninguna técnica específica conocida.

2.3.1 EGSimple (Algortimo Genético Simple)

```
BEGIN
        Generar población inicial
        Calcular el valor fitness de cada individuo
        WHILE NOT Fin DO
        BEGIN
                FOR tamaño de población DO
                BEGIN
                        Selección
                        Cruce
                        Mutación
                        Recalculo fitness
                        Inserción nueva generación
                END
                IF mejor individuo posible alcanzado
                        Terminado := TRUE
        END
END
```

Figura 2-5. Pseudocódigo de un algoritmo genético simple.

Tal y como se vio en la figura 2-2 dónde se expuso el algoritmo genético simple como ejemplo de algoritmo principal, es necesario codificar la solución acorde al tipo de problema. Además de dicha codificación se requiere una función de ajuste que asigne un número real a cada solución codificada, de ello ya hemos hablado durante el epígrafe 2.2.7. Durante la ejecución, se han de seleccionar (epígrafe2.2.2) los padres que elaborarán la reproducción, que se realizará de forma conjunta, primero usando el operador Cruce (epígrafe 2.2.3) generando dos hijos sobre los cuales actuará el operador Mutación (epígrafe 2.2.6). El resultado de los anteriores operadores nos da un nuevo conjunto de individuos, los cuales formarán parte de la siguiente generación de población.

2.3.2 Paralelismo

Durante las pasadas décadas, las mejoras en electrónica permitieron acelerar la capacidad de procesamiento (velocidad). Sin embargo, estamos alcanzando el tope en que la velocidad de los dispositivos electrónicos se aproxima a los límites físicos. De este problema surge el uso del procesamiento en paralelo, el cual nos proporciona medios para continuar el aumento de capacidad de procesamiento.

Para que esto sea posible, el problema debe descomponerse adecuadamente para hacer un uso efectivo de

todos los procesadores. Los datos y el control proporcionan dos áreas principales de la descomposición del problema.

- Con la descomposición de los datos, la misma estructura de control (algoritmo) se coloca en cada procesador y los datos se distribuyen a cada procesador.
- En la descomposición del control, el algoritmo se divide en tareas y las tareas se asignan a diferentes procesadores. Los datos luego fluyen a través de los procesadores como lo requiere el algoritmo.

Los Algoritmos Genéticos utilizan eficientemente (Stender, 1993) las arquitecturas de procesamiento paralelo al usar la descomposición de datos. A continuación, se introducen tres maneras diferentes de explotar el paralelismo de los Algoritmos Genéticos. Ellos son (Lin, Punch, & E.D.Goodman, 1994): Algoritmo Genético paralelo de grano grueso (epígrafe 2.3.2.1), Algoritmo Genético paralelo de micrograno (epígrafe 2.3.2.2) y Algoritmo Genético paralelo de grano fino (epígrafe 2.3.2.3)

2.3.2.1 Algoritmo genético paralelo de grano grueso (o Modelo de islas)

El modelo de islas consiste en hacer una división de la población total en distintas subpoblaciones que se denominan "islas". En cada una de estas islas se ejecuta el Algoritmo Genético y durante cada cierto número de generaciones ocurre un proceso llamado migración que consiste en hacer un movimiento de individuos entre las islas. Ello consigue una explotación de las diferencias endémicas de las distintas subpoblaciones, ganando riqueza en diversidad genética. Es de gran importancia ajustar una tasa concreta de migración que fije la frecuencia con la que va a ocurrir éste fenómeno, cuya importancia es trascendental ya que puede conllevar una convergencia prematura de la búsqueda. En función de la comunicación entre islas podemos distinguir las siguientes modelos (Adbelmalik, Inza, & Larrañaga, 2008):

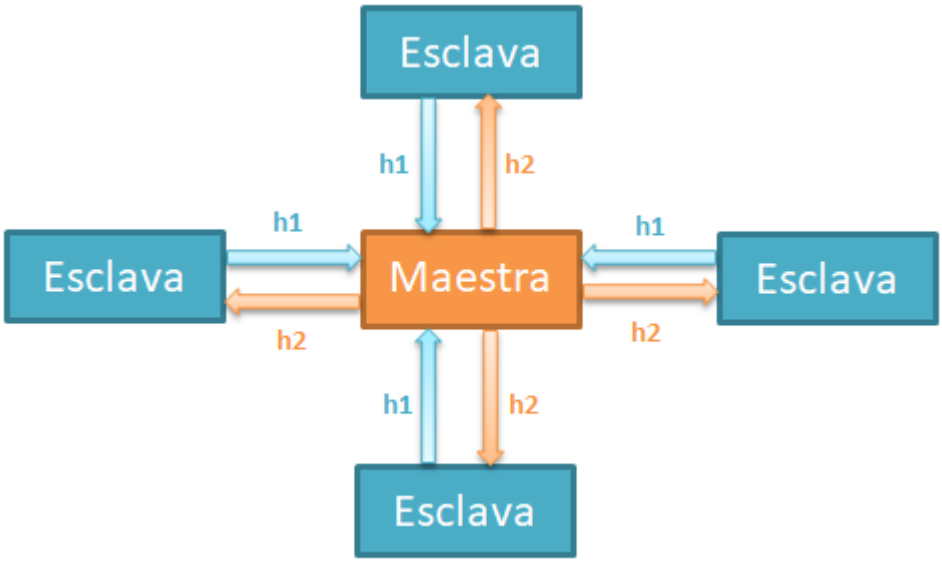

Figura 2-6. Esquema de funcionamiento de modelo de islas con comunicación en estrella.

- Comunicación en estrella: En este tipo de comunicación una de las islas es seleccionada como maestra (generalmente, aquella cuya media de valor fitness es mejor) y las demás esclavas. El modo de comunicarse es el siguiente, todas las islas esclavas mandan sus mejores individuos (h1, variable elegible, pero mayor a uno) a la isla maestra quién recíprocamente envía sus mejores individuos (h2, variable elegible, pero mayor a uno) a cada isla esclava.
- Comunicación en red: En este tipo de comunicación no existe jerarquía alguna, todas las islas mandan sus mejores individuos (h3, variable elegible pero mayor a uno) entre sí a todas las demás islas.
- Comunicación en anillo: En este tipo de comunicación igualmente no existe jerarquía, pero cada isla envía sus mejores individuos (h4, variable elegible pero mayor a uno) únicamente a una población vecina, dotando a la migración con un sentido de flujo.

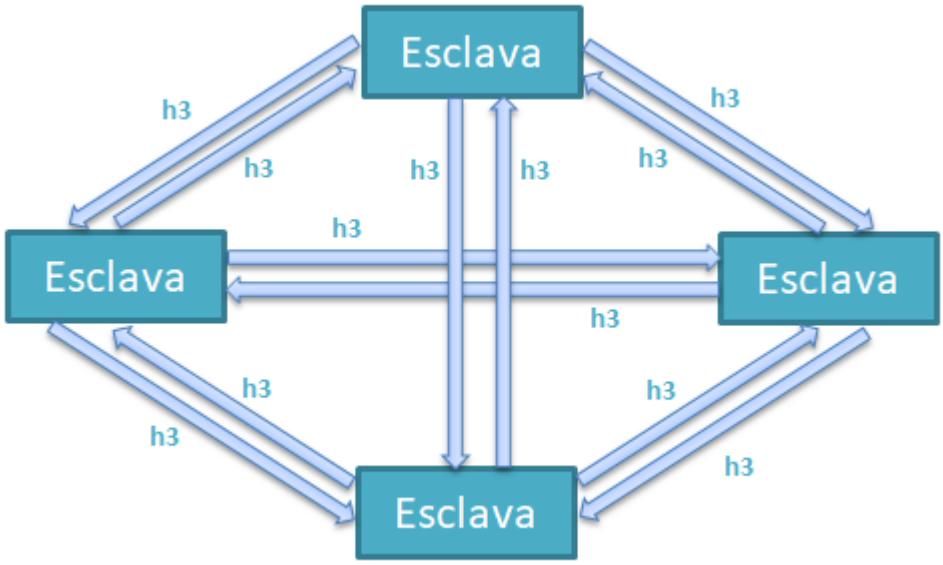

Figura 2-7. Esquema de funcionamiento de modelo de islas con comunicación en red.

Son numerosos los autores que han estudiado y trabajado con los modelos de islas, destacando nombres como (Whitley, The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best., 1989) (Whitley, Beveridge, Guerra, & Graves, 1998) (Starkweather, Whitley, & Mathias, 1991) (Gorges-Schleuter, 1990) (Gordon & Whitley, 1993) (Tomasini, 1999) (Levine, 1993).

Un estudio interesante y detallado sobre el desarrollo de los algoritmos genéticos paralelos puede consultarse en (Nesmachnow, 2002)

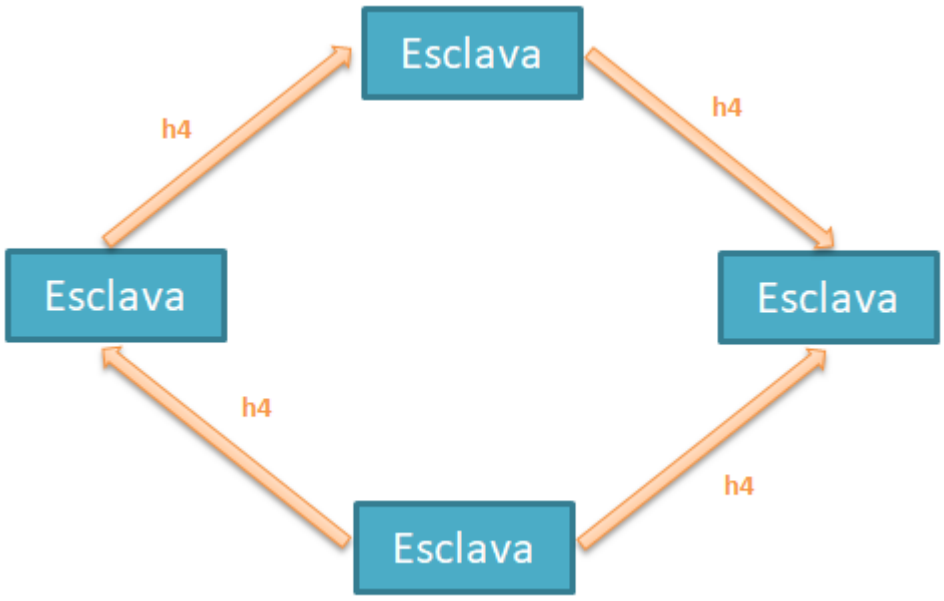

Figura 2-8. Esquema de funcionamiento de modelo de islas con comunicación en anillo.

2.3.2.2 Algoritmo genético paralelo de micrograno

El algoritmo genético paralelo de microgramo es un modelo que trabaja con una única población y hace uso de una comunicación de tipo maestro-esclavo de evaluación de la función de fitness, es decir, un proceso maestro

se encarga de ejecutar los operadores evolutivos (Selección, Cruce, Mutación) y los demás procesos esclavos son los encargados de evaluar la funciones de fitness de cada individuo (Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning., 1989). El rendimiento es superior a un algoritmo genético en serie para complejos pero puede presentar problemas de convergencia prematura.

2.3.2.3 Algoritmo genético paralelo de grano fino (o Modelo Celular)

El algoritmo genético de grano fino es un modelo que asigna a cada elemento de procesamiento (EP) un único individuo y dónde la evaluación se realiza de forma simultáneamente para todos los individuos. La selección, reproducción y cruce se realiza de forma local con un reducido número de vecinos. Esta conectividad entre individuos vecinos aumenta la difusión de individuos aptos, pero convierte a la población susceptible a una convergencia prematura ya que, con el tiempo, se forman grupos homogéneos genéticamente debido a una lenta difusión de individuos. Este modelo celular limita las interacciones, por lo que se ve afectado en el rendimiento.

2.4 El problema de las N Reinas

En este apartado se presenta el problema de las N Reinas, un problema tipo de combinatoria computacional que se va a utilizar para testear diferentes técnicas de paralelización de los algoritmos genéticos basados en islas.

El problema de las N reinas es un problema típico de combinatoria, formulado por primera vez por el ajedrecista Max Bezzel y publicado en Septiembre de 1848 en la revista de ajedrez alemana Schachzeitung (Campbell, 1977) bajo el nombre de: “Problema de las ocho reinas”. Durante este capítulo se estudiará el problema aumentado a un espacio N, es decir, el problema va a consistir en colocar N reinas sobre un tablero cuadrado de ajedrez de NxN subdivisiones, de forma que ninguna reina amenace a otra reina. Una reina amenaza a otra si comparte fila, columna o diagonal con otra reina.

Este tipo de problema tiene una complejidad tal que una búsqueda de todas las posibles combinaciones implicaría un total de N! distintas posibilidades. A modo de ejemplo, en el problema clásico propuesto de ocho reinas, requeriría un tablero 8x8 y por tanto habría un total de 8! = 4320 distintas posibilidades. De todas estas posibilidades tendríamos que quedarnos con aquellas que cumplieran la condición de no amenaza, que son únicamente 92 de ellas. A la vista de este fácil cálculo se deduce que atacar un problema de estas dimensiones con una búsqueda secuencial no tiene sentido y es por ello que vamos a abordarlo a continuación mediante el empleo de Algoritmos Genéticos.

2.4.1 Inicialización. Problema de las N Reinas.

El primer paso para inicializar una población es elegir una codificación adecuada. Para ello, hay que recordar que una de las condiciones de no amenaza es que tanto en cada fila, como en cada columna, ha de existir una única reina. Por lo tanto la codificación que se elige es un vector S = $(X_1, X_2, \dots, X_n)$ tal que cada X_i va a representar una columna en la que se coloca la reina de la fila i.

Al elegir esta representación ya nos aseguramos una de las condiciones de no amenaza, pues habrá una entrada de la tupla por fila, es decir una reina por fila del tablero.

Para asegurar otra de las condiciones de no amenaza, hemos de codificar el problema de forma que los valores de S sean una permutación de la tupla (1,2,…N), de esta forma es seguro que nunca más de una reina estará en una misma columna.

Queda solucionar la última restricción para garantizar que no haya amenaza, dos reinas no han de compartir diagonal. Dos reinas comparten diagonal si para un par de posiciones del vector S = $(X_1, X_2, \dots, X_n)$ se cumple que comparten mismo valor para $(fila - columna)$ o bien para $(fila + columna)$.

2.4.2 Selección. Problema de las N Reinas.

Para el desarrollo de este proyecto, se ha optado por una Selección por torneo probabilístico, con tres individuos distintos como participantes en cada torneo.

2.4.3 Cruce. Problema de las N Reinas.

Ya hemos visto en el epígrafe 2.4.1 que en nuestro algoritmo genético vamos a tener una población cuyos individuos van a ser vectores S = $(X_1, X_2, \dots, X_n)$ dónde cada valor cada X_i será una permutación de la tupla (1,2,…N). Por lo tanto, se han de elegir con especial cuidado las operaciones de cruce y mutación adecuadas ya que se ha de tener en cuenta que cada individuo es una permutación de (1,2,…N) y lo deberá seguir siendo incluso después de aplicar dichos operadores.

El operador cruce que se va a usar es el llamado cruce por emparejamiento parcial (comúnmente conocido como PMX, o Partially Mapped Crossover) cuyo funcionamiento es el siguiente (Larrañaga, Kuijpers, Murga, & Dizdarevic, 1999).

- Se eligen dos puntos de corte aleatorios
- Se copian, en los hijos, las subcadenas comprendida entre dichos puntos de corte, es decir la subcadena del padre uno se copia en la subcadena del hijo dos. La subcadena del padre dos se copia en la subcadena del primer uno.

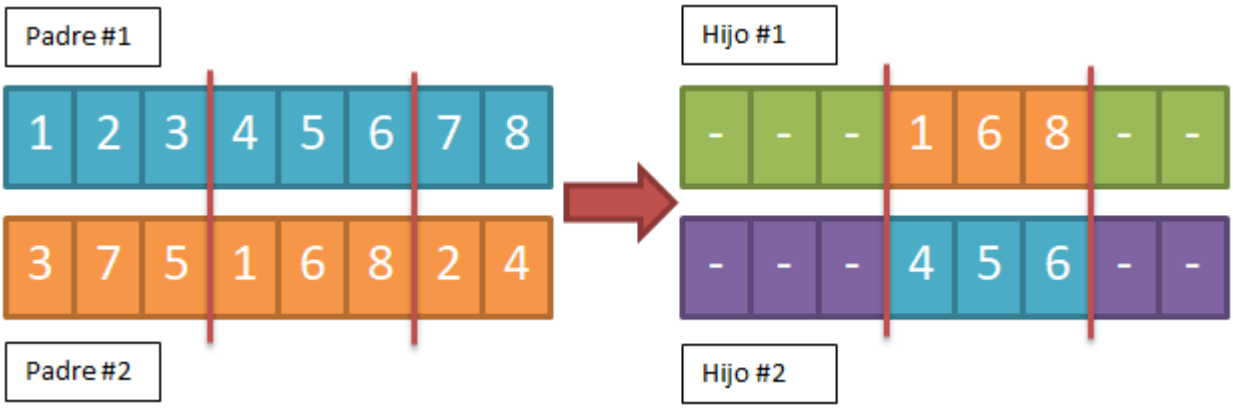

Figura 2-9. Esquema simple representando un ejemplo de cruce por emparejamiento parcial. Primer paso: Copia de la subcadena en la descendencia.

- A continuación se rellena el resto de las celdas de los hijos, de izquierda a derecha. Para ello se siguiendo la siguiente metodología
    - Se copia el valor de la celda del padre i en el hijo i.
    - Si el valor de la celda del padre ya existe en la subcadena que ha sido copiada en el paso anterior se sustituye por el valor asociado al otro hijo. En el ejemplo de la figura 2-10 el primer valor a copiar en el hijo uno sería ‘1’, pero como dicho valor ya ha sido copiado anteriormente, se coge el valor asociado que tenga el hijo dos, que en el caso del ejemplo es ‘4’.

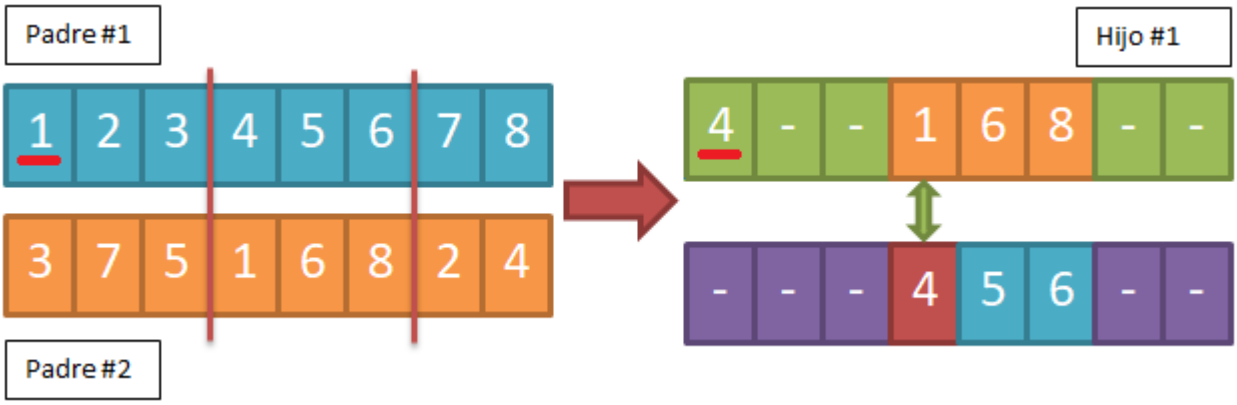

Figura 2-10. Esquema simple representando un ejemplo de cruce por emparejamiento parcial. Segundo paso: Rellenar las demás celdas. Caso en que una celda ya esté en la subcadena central (Celda #1)

- Otro caso que se puede ver en el ejemplo sería el caso de la última celda del hijo uno, que queda recogida en la figura 2-11, ya que las celdas segunda, tercera y séptima se pueden coger sin problema del primer padre. La última celda del padre uno es '8', pero este valor ya aparece dentro del hijo 1 y también lo hace su par con el hijo dos, que sería '6'. En este caso se coge el par de '6' en el hijo uno, que sería '5'.

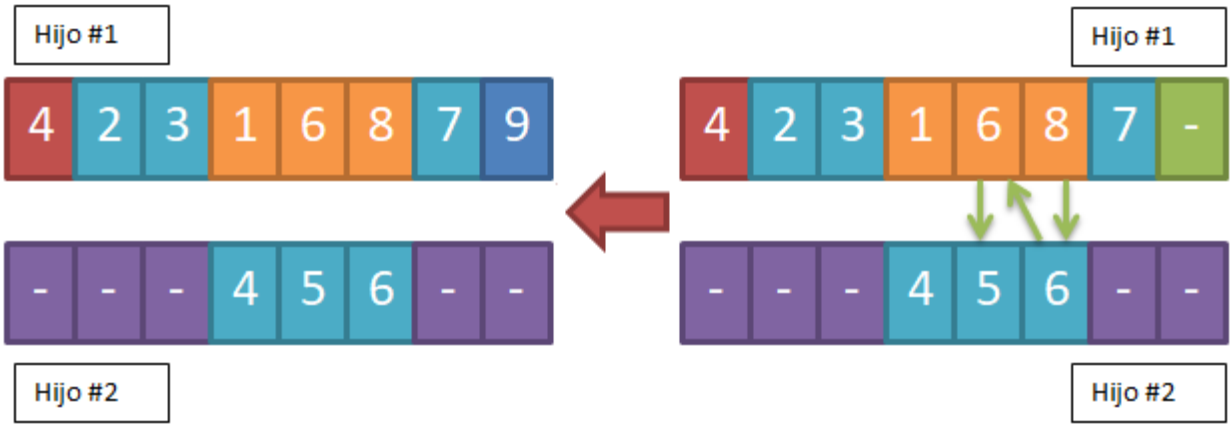

Figura 2-11. Esquema simple representando un ejemplo de cruce por emparejamiento parcial. Segundo paso: Rellenar las demás celdas. Caso en que dos celdas ya estén en la subcadena central. (Celda #8)

- Se procede de nuevo de igual manera con el segundo hijo, pero intercambiado los papeles de los padres.

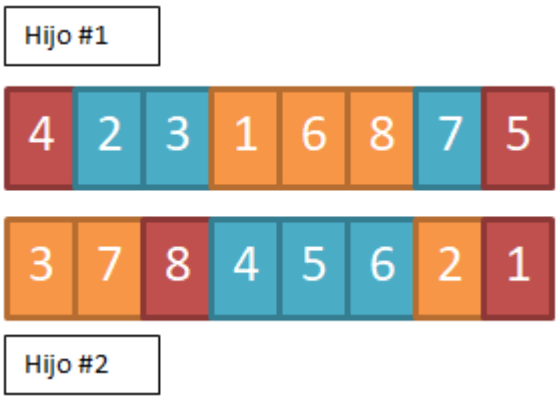

Figura 2-12. Esquema simple representando un ejemplo de cruce por emparejamiento parcial. Tercer paso: Repetir el segundo paso con el segundo hijo. Representación final.

2.4.4 Mutación. Problema de las N Reinas

El operador usado para este problema es muy simple, tan sólo hay que aplicar la mutación como un intercambio entre dos valores y así se mantiene la restricción de formato correctamente formado para el individuo.

2.4.5 Evaluación. Problema de las N Reinas

La función de fitness es la encargada de determinar lo cerca que está cada una de las soluciones de ser una solución válida. Una solución es válida cuando cumple tres condiciones de no ataque, a recordar: Una reina no puede compartir fila con otra reina, condición que se cumple independientemente de la función fitness tal y como se ha codificado el problema (epígrafe 2.4.1), una reina no puede compartir columna con otra reina, condición que se cumple independientemente de la función fitness tal y como se ha codificado el problema y, por último, una reina no puede compartir diagonal con otra reina.

Esta última condición no la hemos abarcado hasta ahora por ningún operador genético y será la función fitness quien va a determinar si una solución es buena, que será si, y sólo si, se cumple esta condición de no compartir diagonal. Tal y como se dijo en el epígrafe 2.4.1 la condición de compartición de diagonal se da si para un par de posiciones del vector S = $(X_1, X_2, \dots, X_n)$ se cumple que comparten mismo valor para (fila-columna) o bien para (fila + columna). Por lo tanto, en la función de evaluación se considera el valor de fitness de una solución como el número de ataques en diagonal. A más conflictos en diagonal peor será la solución y solamente si el número de conflictos es 0, el individuo es solución válida.

3 HARDWARE

El objetivo de este capítulo es describir los dispositivos físicos utilizados en el proyecto, proporcionando descripción, configuración y montaje de los mismos.

3.1 Introducción a Raspberry Pi

Raspberry Pi fue creada por la Raspberry Pi Foundation en 2012 bajo dirección del británico, actual CEO de Raspberry Pi (Trading) Ltd., Eben Upton con la idea original de ser un dispositivo destinado a ser usado con fines de enseñanza y promoción de ciencia básica de computación en escuelas y colegios de Reino Unido (Lyons, 2015). Su popularidad aumentó rápidamente principalmente por su bajo coste, que oscilaba entre las 17 y 23 libras, así como su eficiencia, durabilidad y accesibilidad para modificar y crear proyectos. Se trata además de un dispositivo open hardware (Red Hat, 2010), a excepción del chip primario Broadcomm SoC (siglas para 'System on a Chip'), el cual se encarga de muchas de las funciones principales de la placa – CPU, gráficos, memoria, controlador USB, etc. El software que emplea es, también, open source. Originalmente se crearon 2 imágenes que podrían instalarse fácilmente en una tarjeta SD que luego actuaría como el sistema operativo en el dispositivo, una de ellas se basó en Debian, un popular sistema operativo Linux, y se llamó Raspbian, la otra se llamó RaspBMC basada en Kodi cuyo objetivo era utilizar la Raspberry Pi como un media center.

3.2 Distintos modelos de Raspberry

Desde su fundación en 2012 hasta hoy día Raspberry ha lanzado hasta la tercera versión de Raspberry (Raspberry 3), en distintos modelos, aunque siguen produciendo modelos antiguos ya que por lo general las distintas versiones son compatibles entre sí[1].

3.2.1 Raspberry de primera generación

Originalmente había dos modelos, A y B, con diferentes capacidades y especificaciones.

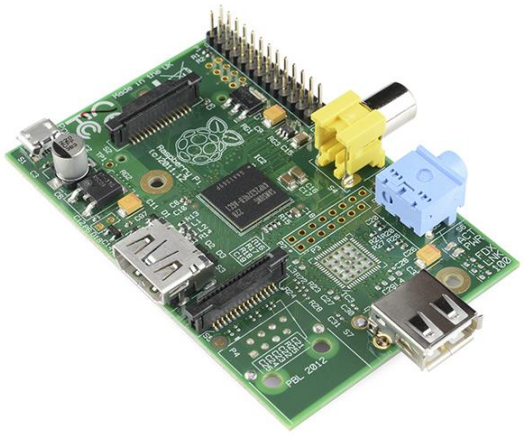

Figura 3-1. Raspberry de primera generación. Modelo A.

[1] A día de hoy, en su página web el modelo más antiguo a la venta es Modelo 1 en sus versiones A+ y B+, no así su primer modelo A y B cuya Entrada/Salida de Propósito General (GPIO) era de tan solo 26 pines y no garantiza compatibilidad con modelos modernos

Raspberry Modelo A tenía una capacidad de 256MB de memoria RAM, un único puerto USB, salida de vídeo vía HDMI (con resoluciones desde 640×350 hasta 1920×1200 con compatibilidad de varios estándares PAL y NTSC) y vídeo RCA. Su precio era inferior al modelo B, así como su consumo.

Raspberry modelo B incluía un Segundo Puerto USB, así como un Puerto Ethernet para conexión a red y 512MB de memoria RAM.

Figura 3-2. Raspberry de primera generación. Modelo B.

Ambos modelos poseen una versión revisada y actualizada, rebautizada como A+ y B+ respectivamente. Presentan mejoras tales como un mayor número de puertos USB, una mejora en el consumo de potencia y lector MicroSDHC en vez de SDHC.

Tabla 1. Comparación entre los distintos modelos Raspberry de primera generación.

	Modelo A	**Modelo A+**	**Modelo B**	**Modelo B+**
Velocidad CPU	700 MHz	700 MHz	700 MHz	700 MHz
Memoria	256mb	256mb	512mb	512mb
Puertos USB	1	1	2	4
GPIO	8	17	8	17
SD/MMC	SD	microSD	SD	microSD
Capacidad de corriente recomendada	700mA	700mA	1.2A	1.8A
Máx. consumo corriente periférica USB	500mA	500mA	500mA	600mA
Corriente activa típica de la placa base	200mA	180mA	500mA	330mA
Tamaño (mm)	85.60 × 56.5	65 × 56.5	85.60 × 56.5	85.60 × 56.5

3.2.2 Raspberry de segunda generación

La segunda generación de Raspberry Pi fue lanzada originalmente en 2015. Ésta versión 2 estrena un procesador ARM Cortex-A7 de cuatro núcleos a diferencia del ARM1176JZF-S de un solo núcleo de su primera versión a una, con lo que se consigue un aumento de la velocidad de procesamiento de 700 MHz en su primera versión a una velocidad de procesamiento de 900MHz. Esta versión mantiene la lectura de MicroSDHC de la versión A+ y B+.

Otras mejoras notorias son la capacidad de memoria RAM, que alcanza 1GB y el número de puertos USB, que aumenta a cuatro.

3.2.3 Raspberry de tercera generación

Raspberry modelo 3 fue lanzado originalmente en 2016, mejora nuevamente el procesador, esta vez un ARM Cortex-A53, que con cuatro núcleos permite alcanzar una velocidad de 1,200MHz. Por primera vez, este modelo incluye de fábrica módulo Wi-Fi 802.11n y módulo Bluetooth 4.1

Durante el año 2018 Raspberry también ha sacado modelos mejorados Raspberry 3 A+ y Raspberry B+.

Raspberry B+ goza del procesador Arm Cortex-A53 de cuatro núcleos que le permite alcanzar velocidad de procesamiento 1.4GH

Raspberry 3 A+ tiene el procesador Arm Cortex-A53 al igual que el modelo B+, pero al igual que el modelo A original solamente tiene 1 puerto USB y no tiene puerto Ethernet. La ausencia de dichos puertos que su forma sea más pequeña y cuadrada en comparación con RPi 3B+.

Sus precios de mercado son de $35 en su versión A+ y $25 en su versión B+.

Tabla 2. Comparación entre los modelos A+, B y B+ de Raspberry de tercera generación.

	Raspberry Pi 3 B	Raspberry Pi 3 B+	Raspberry Pi 3 A+
Velocidad CPU	1.2 GHz	1.4 GHz	1.4 GHz
Memoria	1 GB	1 GB DDR2	512 MB DDR2
Puertos USB	4	4xUSB 2.0	1xUSB 2.0
Ethernet	SI	Gigabit Over USB 2.0	NO
WiFi	802.11n	2.4GHz y 5GHz 802.11 b/g/n/ac	2.4GHz y 5GHz 802.11 b/g/n/ac
GPIO	40	40	40
SD/MMC	microSD	microSD	microSD
Capacidad de corriente recomendada	2.5A	2.5A	2.5A
Máx. consumo corriente periférica USB	1.2A	1.2A	Limitado únicamente por fuente de alimentación, placa y conector.
Corriente activa típica de la placa base	400mA	500mA	350mA
Tamaño (mm)	85.6 × 56.5	85.6 × 56.5	65 x 56

3.3 Material y montaje usado para la realización del proyecto.

Para la realización de este proyecto se ha decidido usar tres dispositivos Raspberry Pi Modelo 3 B, proporcionados por el departamento de Ingeniería Electrónica de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla.

Aprovechando una de las grandes ventajas del modelo 3 de Raspberry, que es la inclusión del módulo Wi-Fi con la placa se ha reducido la complejidad del montaje de las Raspberry Pi y su configuración en red. Por tanto para el montaje del escenario únicamente se ha necesitado de los siguientes periféricos y materiales:

- Una fuente de alimentación Micro USB de, al menos, 5V y cuyo amperaje dependerá de los periféricos que conectemos a la Raspberry: el modelo B consume unos 1200mA en reposo, sin ningún tipo de periférico conectado.
    - Para el desarrollo de este proyecto se ha usado una fuente de 5.1V y 2.5A.
- Únicamente para la configuración y para el mantenimiento se van a hacer uso de otros periféricos, como un ratón y un teclado que únicamente han de conectarse al puerto USB, y una pantalla conectada mediante un cable HDMI para HDTV y haciendo uso del puerto HDMI.
    - Para el desarrollo de este proyecto se ha usado un cable genérico HDMI, un ratón y teclado

inalámbricos genéricos que funcionan con el uso de un solo puerto USB y un monitor HDTV.

- Una tarjeta micro SD para cada Raspberry con una imagen pre-cargada de NOOBS, un asistente de instalación que incluye de fábrica sistema operativo Raspbian, pero que incluye también Pidora, OpenELEC, OSMC, RISC OS, Arch Linux y Windows 10 IOT Core. Pi.

4 SOFTWARE

El objetivo de esta sección es tanto presentar el entorno y librerías con las que se ha trabajado cómo explicar el código desarrollado para la aplicación.

4.1 Configuración de Raspberry

Tal y cómo se ha comentado en el apartado Hardware, hacemos uso de una tarjeta micro SD para cada Raspberry con una imagen pre-cargada de NOOBS, un asistente de instalación que incluye de fábrica sistema operativo Raspbian. No obstante en nuestro caso vamos a hacer uso de una versión modificada de Raspbian, creada por el propio departamento de Ingeniería Electrónica de nuestra escuela.

Figura 4-1. Captura de pantalla del sistema modificado sobre Raspbian.

En esta versión de Raspbian, creado para el Máster universitario en ingeniería electrónica, robótica y automática de la Universidad de Sevilla, tendremos pre-instalados varios entornos de programación tales como Mathematica, BlueJ Jave IDe, Geany, Spyder o Wolfram. En nuestro caso nos interesa Spyder 3, entorno que hemos usado para el desarrollo de los Scripts en Python de los que haremos uso.

Para la instalación de dicha imagen en todas las Raspberry hemos necesitado de un ordenador portátil con lector de tarjetas SD, un adaptador MicroSD – SD, la imagen de RPi2017.iso, proporcionada por el departamento y un software específico Etcher, una aplicación gratuita, de código abierto y multiplataforma que permite crear USBs de arranque de cualquier distribución Linux. Para ello se han seguido los siguientes pasos

- Primero, descargarlo de su página web oficial[2] e instalarlo.
- Obtener un .iso de la imagen a instalar, en este caso RPi2017.iso
- Conectar una tarjeta SD al ordenador con capacidad adecuada para albergar la imagen, en nuestro caso con una tarjeta SD de 16 GB ha sido suficiente.

[2] https://www.balena.io/etcher/

- El último paso es ejecutar el programa. Para *flashear* la imagen en el USB, esto es posible en tan solo tres clics, el primero es para seleccionar la imagen, el segundo para seleccionar la tarjeta SD y el tercero para *flashear*. Una vez hacemos el último click, tras unos minutos el *flasheo* está listo.

Figura 4-2. Captura de pantalla de interfaz de Etcher en un sistema MacOS

En RPi2017, la imagen está pensada para las prácticas de Máster y está optimizado para trabajar tanto en casa como en el entorno de la escuela de ingenieros, esto es, para poder conectarse a internet tanto desde casa como desde la red eduroam, pero evitando pasar por el certificado de seguridad de la red Wi-Fi. Para conseguir esto, por defecto la tarjeta Wi-Fi viene desactivada y está pensado que para acceder a internet, lo haga a través de una conexión permanente RPi – Ordenador personal del alumno haciendo uso de un cable Ethernet, de forma que la configuración de certificados se haga mediante el ordenador y no desde la Raspberry. Pero en nuestro caso esta configuración no nos interesa ya que queremos varias Raspberry Pi interconectadas y descartamos que estén continuamente conectadas vía por Ethernet.

Para habilitar esta configuración, hemos seguido los siguientes pasos en cada una de las RPi.

- Lo primero, habilitar la tarjeta de red Wi-Fi haciendo uso del comando: `sudo ifconfig wlan0 up`
- A continuación se configura una conexión a la red doméstica, para ello se ha hecho uso del comando 'wpa_passphrase' que nos proporciona un archivo de configuración, situado en */etc/wpa_supplicant/wpa_supplicant.conf*, que documenta todas las opciones disponibles y relativas a la configuración. El comando ha sido: `wpa_passphrase nombre-de-la-red < frase-contraseña.txt >`
- Ahora es el turno de configurar la interfaz, para ello se ha modificado el contenido de */etc/network/interfaces*. Aquí, se pueden modficar las distintas interfaces de manera que se le pueda asignar una dirección IP (o usar DHCP, como es el caso), establecer información de ruteo, cambiar máscara IP, asignar rutas por defecto y mucho más. En nuestro caso, hemos añadido las líneas pertinentes para configurar la interfaz wlan0, de manera manual y haciendo uso de un servidor DHCP.

```
autowlan0
        iface wlan0 inet manual
        wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
        iface default inet dhcp
```

- Para hacer efectivos estos cambios, tumbamos y volvemos a levantar la interfaz haciendo uso de los siguientes comandos: `sudo ifup wlan0 ; sudo ifdown wlan0`
- Como se va a necesitar de conexión a internet para descargar las distintas librerías de algoritmos evolutivos y paralelización, hemos de cambiar el Gateway por defecto que, al estar pensado para alumnos de máster, está asignado a la interfaz Ethernet que nosotros no vamos a darle uso. Se han usado los siguientes comandos para realizar dicha acción: `sudo route delete default gw IP Address Adapter` (para eliminar la entrada eth0) y `sudo route add default gw IP Address Adapter` (para añadir la entrada wlan0)

Con esta configuración la Raspberry ya es capaz de conectarse a la red doméstica y tener acceso a internet. Este paso lo repetiremos en todas y cada una de las placas que vamos a usar.

No obstante aún nos queda configurar algo que nos hace falta para poder paralelizar el cómputo. Para la paralelización del algoritmo vamos a necesitar que la carga de trabajo se distribuya en las distintas Raspberry Pi y por tanto vamos a necesitar establecer conexiones por .ssh entre la Raspberry que llamaremos *Broker* y todas las demás, a las cuales nos referimos como *Workers*. Como las conexiones van a ser numerosas, se necesita tener acceso SSH sin restricción de contraseña, es decir, con las siguientes instrucciones conseguimos que cuando queramos paralelizar un proceso nos olvidemos de introducir la clave SSH con el fin de, primeramente, que el usuario no tenga que hacer nada salvo ejecutar el código que lance el proceso y que la máquina no quede tampoco a la espera de la clave SSH.

Para ello se han usado los siguientes comandos, desde la RPi que usamos como *Broker*.

- Lo primero es generar un conjunto de claves de cifrado asimétrico empleando el algoritmo RSA. Para ello se usa el comando: `ssh-keygen -t rsa`
- En en cada una de las Raspberry Workers, se crea un directorio donde almacenar las claves autorizadas para accesos ssh. Esto se consigue con el comando: `ssh pi@192.168.1.49 mkdir -p .ssh` (Siendo 192.168.1.49 la dirección de la Raspberry Broker)
- Se pasa la clave de cifrado que hemos creado en el paso uno a las distintas RPi, en el directorio que acabamos de crear, y se les da los permisos adecuados. Ello se consigue con los comandos: `cat .ssh/id_rsa.pub | ssh pi@192.168.1.49 'cat >> .ssh/authorized_keys'` para copiar la clave cifrada al worker y `ssh pi@192.168.1.49 "chmod 700 .ssh; chmod 640 .ssh/authorized_keys"` para dar los permisos.
- Por último, hacer simplemente una prueba de que desde la RPi Broker podemos acceder por ssh a cualquier otra, sin necesidad de introducir clave alguna mediante `ssh pi@192.168.1.X` (donde x es distinto cada Raspberry worker)

Para poder ejecutar los diferentes scripts que vamos a lanzar, necesitamos obviamente Python, que no nos hemos tenido que preocupar pues la imagen proporcionada por la escuela, RPi2017.iso, ya incluye paquete Python. Pero vamos a necesitar también dos librerías de terceros, DEAP y SCOOP.

4.2 Librería DEAP (Distributed Evolutive Algorithm Python)

DEAP es un framework de computación evolutiva que busca hacer que los algoritmos y estructuras de datos sean transparentes[3]. Una de sus grandes ventajas y razón por la cuál ha sido elegido para el desarrollo de este trabajo es que funciona en perfecta armonía con mecanismos de paralelización como son mutiprocessing o SCOOP.

Las principales diferencias entre DEAP y otros frameworks de algoritmos evolutivos es que en lugar de trabajar con Tipos predefinidos, proporciona formas de crear tipos nuevos, igual ocurre con Inicializadores, permitiendo iniciadores personalizarlos en lugar de predefinidos. También en lugar de implementar algoritmos cerrados, DEAP permite escribir uno adecuado a cada caso.

- Tipos: Python permite crear distintos *Tipos* haciendo uso del módulo Creator, una meta-fábrica que permite crear clases de distinto tipo que satisfarán las necesidades de los algoritmos evolutivos.
- Iniciadores: Una vez han sido creados los tipos, han de rellenarse con valores, a veces, aleatorios y, otras, designados. DEAP proporciona un mecanismo para facilitar precisamente esto. *Toolbox* es un contenedor para herramientas de todo tipo, incluidos los inicializadores.
- Operadores: Los operadores son como inicializadores, excepto que algunos ya están implementados en el módulo *tools*. Una vez que hayan sido elegidos únicamente hay que registrarlo en la *toolbox*. La función de evaluación, en DEAP, hay que hacerla de esta manera.
- Algoritmos: Normalmente es una función tipo main(), también es posible usar uno de los cuatro algoritmos disponibles en el módulo *algorithms.* Estos son:
    - eaSimple
    - eaMuPlusLambda
    - eaMuCommaLambda
    - eaGenerateUpdate

4.2.1 Instalación

Simplemente hacer uso del comando: `pip install deap`

4.3 Librería SCOOP (Scalable COncurrent Operations in Python)

SCOOP es un módulo de tareas distribuidas que permite programación paralela concurrente en varios entornos, desde mallas heterogéneas hasta supercomputadoras. Utiliza tanto ZeroMQ, una librería que permite intercambio de mensajes asíncronos con alto rendimiento y destinada al uso en aplicaciones distribuidas o concurrentes, como Greenlet, un paquete que permite trabajar con estructuras livianas similares a hilos que se programan y administran dentro del proceso. Tanto ZeroMQ, como Greenlet son usados para encapsular y distribuir tareas (llamadas *Future*) entre procesos y/o sistemas.

SCOOP fue diseñado a partir de las siguientes ideas:

- El futuro es paralelo;
- Lo simple es bonito;
- El paralelismo debería ser más simple.

Y a su vez, sus características son las siguientes

- Aprovechar el poder de computación de varias máquinas haciendo uso en red.
- Capacidad de generar múltiples tareas dentro de una tarea.
- API compatible con PEP-3148.

[3] https://deap.readthedocs.io/en/master/

- Paralelizar código secuencial con sólo pequeñas modificaciones;
- Equilibrio de carga eficiente.

El módulo SCOOP genera los Broker y Workers necesarios en una lista dada de máquinas, incluidas máquinas remotas a las cuales se comunican a través de ssh. Se puede especificar los hosts con un archivo hostfile y pasarlo a SCOOP usando el argumento --hostfile. El archivo hostfile debe seguir una sintaxis fijada.

4.3.1 Instalación

Por supuesto, SCOOP necesita como requisito indispensable tener previamente instalado Python, 2.6 ó >= 3.2. En nuestro caso, Python 2.7.13 ya está instalado en RPi2017, por lo que esta dependencia no nos afecta.

La lista completa de dependencias es la siguiente

- Python >= 2.6 or >= 3.2
- Distribute >= 0.6.2 or setuptools >= 0.7
- Greenlet >= 0.3.4
- pyzmq >= 13.1.0 and libzmq >= 3.2.0
- ssh for remote execution

Para instalar Scoop únicamente ha hecho falta hacer uso del comando: `pip install scoop`

4.4 Python

Creado por el holandés Guido van Rossum y pensado como sucesor del lenguaje de programación ABC (Nosuna, 2011), Python es un lenguaje de programación interpretado con hincapié en una sintaxis limpia y que favorezca un código legible, similar al pseudocódigo utilizado para esquematizar la programación.

Las características que marcan a Python son:

- Ser tanto una herramienta potente como intuitiva
- Que cualquiera pueda contribuir a su desarrollo (Open source)
- Tan legible y entendible como el inglés.
- Que sea de apropiado uso para tareas diarias, es decir, que permita tiempos cortos de desarrollo.

Además, en torno a la comunidad creada en Python se ha creado lo que se conoce como 'Filosofía Python' una serie de principios de legibilidad y transparencia, en las que se destaca, entre otras:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.

4.4.1 ¿Por qué Python?

Python es una herramienta que permite realizar una programación completa y por ello se le denomina lenguaje de programación multiparadigma, ya que permite tanto programación con diseño orientado a objetos, unidades de testeo, generación de documentación e incluso alguna interacción con el sistema operativo. Actualemente Python tiene un amplio conjunto de librerías que extiende su funcionalidad en el ámbito científico, incluyendo

tareas tanto de tratamiento de datos, como visualización, cálculo numérico y simbólico entre otras aplicaciones específicas. Todo ello arropado por una comunidad de usuarios que, debido a la filosofía código abierto, son proclives a compartir su código y mejorarlo entre todos.

5 METODOLOGÍA

Para llevar a cabo los objetivos planteados en este proyecto se ha seguido la siguiente metodología a partir del diseño del experimento que se describe:

Lo primero ha sido llevar a cabo tanto el montaje de los dispositivos Raspberry, como su posterior configuración conforme a lo descrito en el epígrafe 3.3.

Lo siguiente ha consistido en la implementación del código necesario para realizar ensayos de forma que solucionemos el problema de las N Reinas, desarrollado en el epígrafe 2.4, de las siguientes formas:

1. Haciendo uso de algoritmo genético simple que nos proporciona la librería DEAP, en una sola máquina y de forma secuencial. Con una única población total.
2. Haciendo uso de algoritmo genético simple que nos proporciona la librería DEAP y haciendo uso de la paralelización que nos proporciona la librería SCOOP, de nuevo con una única población total.
3. Haciendo uso de algoritmo genético simple que nos proporciona la librería DEAP e implementando un modelo de islas de forma que se va a dividir la población total en diversas subpoblaciones y estableciendo un flujo de migración entre las mismas.
4. Haciendo uso de algoritmo genético simple que nos proporciona la librería DEAP e implementando un modelo de islas de forma que se divida la población total en diversas subpoblaciones, estableciendo un flujo de migración entre las mismas y haciendo uso de las capacidades de la librería SCOOP para paralelizar la carga de trabajo entre varios workers.

A continuación, se ha planteado una serie de ensayos consistentes en la resolución de cinco escenarios, que hemos llamado A, B, C, D y E, tales que:

- Escenario A: se resuelve el problema de las N Reinas bajo la condición de que el tablero tenga unas dimensiones 20x20 y que el algoritmo genético va a trabajar con una población total de 300 individuos, una probabilidad de cruce de 0.5 y una probabilidad de mutación de 0.2.
- Escenario B: se resuelve el problema de las N Reinas bajo la condición de que el tablero tenga unas dimensiones 40x40 y que el algoritmo genético va a trabajar con una población total de 300 individuos, una probabilidad de cruce de 0.5 y una probabilidad de mutación de 0.2.
- Escenario C: se resuelve el problema de las N Reinas bajo la condición de que el tablero tenga unas dimensiones 60x60 y que el algoritmo genético va a trabajar con una población total de 300 individuos, una probabilidad de cruce de 0.5 y una probabilidad de mutación de 0.2.
- Escenario D: se resuelve el problema de las N Reinas bajo la condición de que el tablero tenga unas dimensiones 80x80 y que el algoritmo genético va a trabajar con una población total de 300 individuos, una probabilidad de cruce de 0.5 y una probabilidad de mutación de 0.2.
- Escenario E: se resuelve el problema de las N Reinas bajo la condición de que el tablero tenga unas dimensiones 100x100 y que el algoritmo genético va a trabajar con una población total de 300 individuos, una probabilidad de cruce de 0.5 y una probabilidad de mutación de 0.2.

Estos escenarios van a resolverse en los siguientes casos (Tabla 3)

1. Un ensayo secuencial sin SCOOP: Como su nombre indica no se va a hacer uso de la librería SCOOP, por lo tanto se va a resolver el problema de las N Reinas usando un algoritmo evolutivo simple sin paralelización de carga. Esto implica que únicamente vamos a hacer uso de un dispositivo Raspberry Pi.
2. Un ensayo haciendo uso del modelo de islas, pero sin hacer uso de SCOOP: La población total, 300 en todos los escenarios, se dividirá en subpoblaciones asignadas a cada isla, 3 en todos los escenarios, por lo tanto cada isla contará con un total de 100 individuos. No obstante, en este caso no paralelizaremos el algoritmo, por lo tanto será un único ensayo en un dispositivo Raspberry Pi.
3. Seis ensayo secuenciales haciendo uso de SCOOP. En esta serie de ensayos se va a hacer uso tanto de

la librería DEAP como SCOOP, por lo tanto todos ellos van a paralelizar la carga. En estos ensayos se va a usar tanto una, dos y tres dispositivos Raspberry Pi. En el caso de una única Raspberry no distribuiremos la carga en red porque el experimento será en local. En todos estos casos vamos a realizar el ensayo en dos vertientes, distribuyendo la carga en dos workers por dispositivo RPi o en cuatro.

4. Seis ensayos haciendo uso del modelo de islas y SCOOP: La población total, 300 en todos los escenarios, se dividirá en subpoblaciones asignadas a cada isla, 3 en todos los escenarios, por lo tanto cada isla contará con un total de 100 individuos. Nuevamente, en estos ensayos se va a usar tanto una, dos y tres dispositivos Raspberry Pi. En el caso de una única Raspberry no distribuiremos la carga en red porque el experimento será en local. En todos estos casos vamos a realizar el ensayo en dos vertientes, distribuyendo la carga en dos workers por dispositivo RPi o en cuatro.

Estos 14 casos planteados se van a llevar a cabo en los cinco escenarios, descritos anteriormente y realizando un total de 70 casos. Puesto que el tiempo de ejecución y los resultados obtenidos varían en cada situación de forma impredecible, se ha optado que por cada uno de estos 70 ensayos se repita 5 veces, elevando el número de ejecuciones a 350. Para el posterior análisis de resultados se han recogido los valores medios de cada una de estas cinco repeticiones.

Tabla 3. Ensayos realizados por escenario

Modelo AG	Paralelo	Distribuido	Nº RPi	Workers
Modelo Secuencial sin Scoop	NO	NO	1	
Modelo Secuencial con Scoop	SI	NO	1	2
				4
	SI	SI	2	2
				4
			3	2
				4
Modelo de islas sin Scoop	NO	NO	1	
Modelo de islas con Scoop	SI	NO	1	2
				4
	SI	SI	2	2
				4
			3	2
				4

5.1 Código escrito

Para la realización de este trabajo se ha realizado una batería de pruebas en los que se hacen uso de cuatro scripts escritos en Python que representan las cuatro técnicas que van a ser evaluadas y en los que, con el fin de adecuarse a los diferentes escenarios, aceptan como entrada los diferentes parámetros que formarán el algoritmo genético. Para los casos que usen SCOOP se han escrito ficheros hostfile que se pasan a la librería y sirven para designar los Brokers y Workers que se van a desplegar. Como se comentó en la introducción de este capítulo por cada escenario se van a resolver 14 casos, cada uno repetido cinco veces. Para automatizar este proceso, se han realizado diferentes scripts en shellscript. Además, se ha diseño un script más en Python que calcula las medias de estas cinco repeticiones por caso.

5.1.1 Scripts en Python

- ***Ensayo_Sec_NoScoop.py*:** Script que acepta por parámetro de entrada la **dimensión N** del tablero, el **número de generaciones** que van a ser evaluadas, el **tamaño de la población** generacional, el **nombre del fichero** donde se van a guardar los resultados, **la probabilidad de cruce** y la **probabilidad de mutación**. La peculiaridad de este script respecto al siguiente que se describe a continuación es que no hace uso de la librería SCOOP y por lo tanto su ejecución se realiza secuencialmente, de forma local en la misma máquina en la que se ejecuta. En el fichero de resultados se anota un array con las generaciones que se han muestreado, un array con el mejor valor encontrado en cada generación y finalmente el tiempo de ejecución total.
- ***Ensayo_Sec_Scoop.py:*** Script que acepta por parámetro de entrada la **dimensión N** del tablero, el **número de generaciones** que van a ser evaluadas, el **tamaño de la población** generacional, el **nombre del fichero** donde se van a guardar los resultados, **la probabilidad de cruce** y la **probabilidad de mutación**. Para ejecutar este Script que, como su propio nombre indica, hará uso de paralelización de trabajo mediante SCOOP se ha de señalizar la opción -- hosfile por línea de comandos, tras la cual le añadiremos el fichero hostfile personalizado con las direcciones IP de las distintas RPi de las que haremos uso y en las cuales distribuiremos la carga de la ejecución. En el fichero de resultados se anota un array con las generaciones que se han muestreado, un array con el mejor valor encontrado en cada generación y finalmente el tiempo de total.
- ***Ensayo_Islas_NoScoop.py:*** Script que acepta por parámetro de entrada la **dimensión N** del tablero, el **número de generaciones** que van a ser evaluadas, el **tamaño de la población** generacional, el **nombre del fichero** donde se van a guardar los resultados, el **número de islas** en el que se va a dividir la población, la **frecuencia** en que va a ocurrir la migración, el número **de individuos que van a migrar** por cada migración, **la probabilidad de cruce** y la **probabilidad de mutación**. La peculiaridad de este script respecto al siguiente que se describe a continuación es que no hace uso de la librería SCOOP y por lo tanto su ejecución se realiza secuencialmente, de forma local en la misma máquina en la que se ejecuta. En el fichero de resultados se anota un array con las generaciones que se han muestreado, que en este caso será una muestra por migración, un array con el mejor valor encontrado en cada generación y finalmente el tiempo de total.
- ***Ensayo_Islas_NoScoop.py:*** Script que acepta por parámetro de entrada la **dimensión N** del tablero, el **número de generaciones** que van a ser evaluadas, el **tamaño de la población** generacional, el **nombre del fichero** donde se van a guardar los resultados, el **número de islas** en el que se va a dividir la población, la **frecuencia** en que va a ocurrir la migración, el número **de individuos que van a migrar** por cada migración, **la probabilidad de cruce** y la **probabilidad de mutación**. Nuevamente para la ejecución de este Script se hará uso de la opción -- hostfile, especificando el fichero que contiene las IPs de las RPi en las que se va a distribuir la carga junto al número de workers que operarán. En el fichero de resultados se anota un array con las generaciones que se han muestreado, que en este caso será una muestra por migración, un array con el mejor valor encontrado en cada generación y finalmente el tiempo de total.
- ***HacerDatos.py:*** Este sencillo Script acepta dos parámetros de entrada: El nombre un archivo de resultados del que va a obtener datos y el nombre de un archivo nuevo a crear y en el que se van a guardar las medias del contenido del primer archivo. Por lo tanto en este script se calcula la media,

generación a generación, de los valores de fitness de todos los ensayos guardados en el archivo y también se calcula la media de los tiempos de cada ensayo y su desviación típica.

```
gen:15,30,45,60,75,90,105,120,135,150,165,180,195,210,225,240,
255,270,285,300,
fit:16.0,6.0,3.0,3.0,2.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1
.0,1.0,1.0,1.0,1.0,
Elapsed time:123.505956888
---
gen:15,30,45,60,75,90,105,120,135,150,165,180,195,210,225,240,
255,270,285,300,
fit:13.0,10.0,8.0,8.0,4.0,4.0,3.0,3.0,2.0,2.0,2.0,2.0,2.0,2.0,
2.0,2.0,2.0,2.0,2.0,2.0,
Elapsed time:121.246194839
---
gen:15,30,45,60,75,90,105,120,135,150,165,180,195,210,225,240,
255,270,285,300,
fit:13.0,8.0,8.0,4.0,4.0,3.0,3.0,3.0,3.0,2.0,1.0,1.0,0.0,0.0,0
.0,0.0,0.0,0.0,0.0,0.0,
Elapsed time:125.186691046
---
gen:15,30,45,60,75,90,105,120,135,150,165,180,195,210,225,240,
```

Figura 5-1 Ejemplo de archivo de resultados generado por Ensayo_Isla_NoScoop.py

5.1.2 Scripts en shell

Para la realización de todos los casos descritos en la introducción de este capítulo, se ha realizado una batería de scripts con las siguientes características.

- **Script_Sec_NoScoop_A.sh, Script_ Sec_NoScoop_B.sh, Script_ Sec_NoScoop_C.sh, Script_ Sec_NoScoop_D.sh, Script_ Sec_NoScoop_E.sh:** Este ensayo se realiza en local, es decir no se necesita de ningún fichero hostfile. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Sec_NoScoop.py* con las siguientes especificaciones según el escenario:
    - **Script_Sec_NoScoop_A**) Un tablero 20x20, a resolver en 100 generaciones con una población total de 300 individuos. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
    - **Script_Sec_NoScoop_B**) Un tablero 40x40, a resolver en 200 generaciones con una población total de 300 individuos. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
    - **Script_Sec_NoScoop_C**) Un tablero 60x60, a resolver en 300 generaciones con una población total de 300 individuos. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
    - **Script_Sec_NoScoop_D**) Un tablero 80x80, a resolver en 400 generaciones con una población total de 300 individuos. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
    - **Script_Sec_NoScoop_E**) Un tablero 100x100, a resolver en 500 generaciones con una población total de 300 individuos. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
- **Script_Islas_NoScoop_A.sh, Script_ Islas _NoScoop_B.sh, Script_ Islas _NoScoop_C.sh, Script_ Islas _NoScoop_D.sh, Script_ Islas _NoScoop_E.sh:** Este ensayo se realiza en local, es decir no se necesita de ningún fichero hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Isla_NoScoop.py* con las siguientes especificaciones según el escenario:

  - **Script_Isla_NoScoop_A**) Un tablero 20x20, en 100 generaciones con una población de 3 islas con 100 individuos cada una. La frecuencia de migración será cada 5 generaciones y se intercambiará un volumen de 15 individuos por isla en cada migración. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
  - **Script_Isla_NoScoop_B**) Un tablero 40x40, en 200 generaciones con una población de 3 islas con 100 individuos cada una. La frecuencia de migración será cada 10 generaciones y se intercambiará un volumen de 15 individuos por isla en cada migración. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
  - **Script_Isla_NoScoop_C**) Un tablero 60x60, en 300 generaciones con una población de 3 islas con 100 individuos cada una. La frecuencia de migración será cada 15 generaciones y se intercambiará un volumen de 15 individuos por isla en cada migración. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
  - **Script_Isla_NoScoop_D**) Un tablero 80x80, en 300 generaciones con una población de 3 islas con 100 individuos cada una. La frecuencia de migración será cada 20 generaciones y se intercambiará un volumen de 15 individuos por isla en cada migración. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
  - **Script_Isla_NoScoop_E**) Un tablero 100x100, en 300 generaciones con una población de 3 islas con 100 individuos cada una. La frecuencia de migración será cada 25 generaciones y se intercambiará un volumen de 15 individuos por isla en cada migración. La probabilidad de cruce especificada para todo el problema es 0.5 para cruce y 0.2 para mutación.
- **Script_Sec_Scoop_A_1rpi.sh, Script_Sec_Scoop_B_1rpi.sh, Script_Sec_Scoop_C_1rpi.sh, Script_Sec_Scoop_D_1rpi.sh, Script_Sec_Scoop_E_1rpi.sh:** Este ensayo se realiza de forma distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Sec_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Sec_NoScoop_A/ Script_Sec_NoScoop_B/ Script_Sec_NoScoop_C/ Script_Sec_NoScoop_D/ Script_Sec_NoScoop_E. Los archivos de resultados van a anotar lo siguiente:
  - Resultados_Sec_Scoop_A_1RPI_2Wor: En este archivo de resultados se anota la salida cuando el fichero hostfile especifica una única máquina con dos workers. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
  - Resultados_Sec_Scoop_A_1RPI_4Wor: En este archivo de resultados se anota la salida cuando el fichero hostfile especifica una única máquina con cuatro workers. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
- **Script_Sec_Scoop_A_2rpi.sh, Script_Sec_Scoop_B_2rpi.sh, Script_Sec_Scoop_C_2rpi.sh, Script_Sec_Scoop_D_2rpi.sh, Script_Sec_Scoop_E_2rpi.sh:** Este ensayo se realiza de forma distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Sec_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Sec_NoScoop_A/ Script_Sec_NoScoop_B/ Script_Sec_NoScoop_C/ Script_Sec_NoScoop_D/ Script_Sec_NoScoop_E. Los archivos de resultados van a anotar lo siguiente:
  - Resultados_Sec_Scoop_A_2RPI_2Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica dos máquinas con dos workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
  - Resultados_Sec_Scoop_A_2RPI_4Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica dos máquinas con cuatro workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
- **Script_Sec_Scoop_A_3rpi.sh, Script_Sec_Scoop_B_3rpi.sh, Script_Sec_Scoop_C_3rpi.sh, Script_Sec_Scoop_D_3rpi.sh, Script_Sec_Scoop_E_3rpi.sh:** Este ensayo se realiza de forma

distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Sec_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Sec_NoScoop_A/ Script_Sec_NoScoop_B/ Script_Sec_NoScoop_C/ Script_Sec_NoScoop_D/ Script_Sec_NoScoop_E. Los archivos de resultados van a anotar lo siguiente:

  - Resultados_Sec_Scoop_A_3RPI_2Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica tres máquinas con dos workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
  - Resultados_Sec_Scoop_A_3RPI_4Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica tres máquinas con dos workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.

- **Script_Isla_Scoop_A_1rpi.sh, Script_Isla_Scoop_B_1rpi.sh, Script_Isla_Scoop_C_1rpi.sh, Script_Isla_Scoop_D_1rpi.sh, Script_Isla_Scoop_E_1rpi.sh:** Este ensayo se realiza de forma distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Isla_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Isla_NoScoop_A/ Script_Isla_NoScoop_B/ Script_Isla_NoScoop_C/ Script_Isla_NoScoop_D/ Script_Isla_NoScoop_E. Los archivos de resultados van a anotar las siguientes especificaciones.
  - Resultados_Isla_Scoop_A_1RPI_2Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica una única máquina con dos workers. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
  - Resultados_Ensayo_IslaBest_A_1RPI_4Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica una única máquina con cuatro workers. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
- **Script_Isla_Scoop_A_2rpi.sh, Script_Isla_Scoop_B_2rpi.sh, Script_Isla_Scoop_C_2rpi.sh, Script_Isla_Scoop_D_2rpi.sh, Script_Isla_Scoop_E_2rpi.sh:** Este ensayo se realiza de forma distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Isla_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Isla_NoScoop_A/ Script_Isla_NoScoop_B/ Script_Isla_NoScoop_C/ Script_Isla_NoScoop_D/ Script_Isla_NoScoop_E. Los archivos de resultados van a anotar las siguientes especificaciones.
  - Resultados_Isla_Scoop_A_2RPI_2Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica dos máquinas con dos workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
  - Resultados_Ensayo_IslaBest_A_2RPI_4Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica dos máquinas con cuatro workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.
- **Script_Isla_Scoop_A_3rpi.sh, Script_Isla_Scoop_B_3rpi.sh, Script_Isla_Scoop_C_3rpi.sh, Script_Isla_Scoop_D_3rpi.sh, Script_Isla_Scoop_E_3rpi.sh:** Este ensayo se realiza de forma distribuida, por lo tanto sí necesitará de ficheros hostfile para ejecutarlo. Se trata de una iteración dónde se va a ejecutar 5 veces el script *Ensayo_Isla_Scoop.py* con los mismos parámetros, según el escenario que corresponda, que los especificados en el fichero Script_Isla_NoScoop_A/ Script_Isla_NoScoop_B/ Script_Isla_NoScoop_C/ Script_Isla_NoScoop_D/ Script_Isla_NoScoop_E. Los archivos de resultados van a anotar las siguientes especificaciones.
  - Resultados_Isla_Scoop_A_3RPI_2Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica tres máquinas con dos workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.

- Resultados_Ensayo_IslaBest_A_3RPI_4Wor: En este archivo de resultados se ha anotado la salida cuando el fichero hostfile especifica tres máquinas con cuatro workers cada una. De forma análoga, los escenarios B, C, D y E generarán ficheros de resultados similares.

5.1.3 Ficheros hostfile

Para poder realizar los distintos ensayos de manera distribuida haciendo uso de SCOOP se le ha de proporcionar mediante el comando --hostfile un archivo con una lista de los workers que se van a desplegar. A continuación, se listan los distintos ficheros hostfile de los que se han hecho uso y su contenido.

- hostfile_1rpi_2worker

  192.168.1.49 2

- hostfile_1rpi_4worker:

  192.168.1.49 4

- hostfile_2rpi_2worker:

  192.168.1.49 2

  192.168.1.51 2

- hostfile_2rpi_4worker:

  192.168.1.49 4

  192.168.1.51 4

- hostfile_3rpi_2worker:

  192.168.1.49 4

  192.168.1.51 4

  192.168.1.50 4

- hostfile_3rpi_4worker:

  192.168.1.49 4

  192.168.1.51 4

  192.168.1.50 4

6 RESULTADOS

En este capitulo se presenta un análisis de los resultados recogidos tras la ejecución de los casos descritos en el capítulo anterior.

Comencemos por el escenario A. Recordemos, este escenario cuenta con un array de solución de tan sólo 20 entradas (ya que el tablero es 20x20), y una población total de 300 individuos así como un tiempo generacional de 100 generaciones. De la tabla 4,5 y 6 podemos concluir con algunas consideraciones importantes.

- Una de ellas sería que no existe una diferencia apreciable en términos de tiempo medio de ejecución (sólo un 5.26% de incremento), en el caso de no usar SCOOP, entre usar un modelo de población total o un modelo de subpoblaciones con migración.
- Un resultado importante (y que se va a dar en todos los escenarios) es que aunque tengamos 2 o 4 workers por dispositivo e independientemente de tener una, dos o tres Raspberry Pi puestas a trabajar, en el caso de usar un algoritmo genético de población total con SCOOP, el tiempo no varía en gran medida. En este escenario se consigue, apenas, una reducción de tiempo de ejecución del 9.62% entre los dos casos más extremos, 1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo.
- Otra consideración a destacar es, si se observa en las filas de método con islas y SCOOP, que conforme se va distribuyendo en más workers, el tiempo medio de ejecución se va reduciendo. Lo cual quiere decir que al contrario que con el modelo secuencial, aquí SCOOP tiene algo más para paralelizar. Al dividir la población total en subpoblaciones casi independientes, estamos dando más libertad a la paralelización y explotamos más eficazmente las prestaciones de SCOOP. En este escenario se consigue una reducción en el tiempo de ejecución en un 42.42% entre los casos más extremos, 1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo.
- Finalmente, observar que en el caso de ejecutar el algoritmo con modelo de islas sin paralelización obtenemos un tiempo medio de unos 20 segundos por ejecución mientras que si nos vamos al extremo opuesto, a la máxima distribución (3 Raspberry Pi a 4 Worker por dispositivo, modelo de islas paralelizado con SCOOP), observamos un tiempo medio de unos 38 segundos. Es decir se emplea un 90% más de tiempo que si no hubiéramos usado la librería SCOOP.
    - De manera análoga si comparamos entre ejecutar el algoritmo de población total sin paralelización frente al modelo de población total con SCOOP, obtenemos que se emplea un 168.42% más de tiempo en este segundo modelo.

Método	1 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	19.74775	0.60309	-	-	-	-
Secuencial (Scoop)	-	-	57.84502	1.03282	52.92366	0.48318
Islas (No Scoop)	20.43486	0.38390	-	-	-	-
Islas (Scoop)	-	-	47.32385	0.61884	38.71383	0.43921

Tabla 4. Tabla de tiempos de ejecución, en segundos, de Escenario A. 1 Dispositivo Raspberry.

Método	2 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	52.79206	0.50829	53.00013	1.38337
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	37.14940	0.92034	35.98973	0.41360

Tabla 5. Tabla de tiempos de ejecución, en segundos, de Escenario A. 2 Dispositivos Raspberry.

Método	3 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	51.83221	0.68042	52.27083	0.52572
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	36.12599	0.74116	33.77954	1.07517

Tabla 6. Tabla de tiempos de ejecución, en segundos, de Escenario A. 3 Dispositivos Raspberry.

Con respecto a las tablas 7,8 y 9 correspondientes al escenario B, que recordemos cuenta con un array de solución 40 entradas (ya que el tablero es 40x40), una población total de 300 individuos y se resuelve en 200 generaciones, cabe señalar que:

- Al igual que observamos en el escenario A, en el caso de no usar SCOOP, el tiempo no varía en gran medida (sólo un 1.72% de incremento) entre usar un modelo de población total o un modelo de subpoblaciones con migración.
- Nuevamente, en el caso de usar un algoritmo genético de población total paralelizado con SCOOP el tiempo no disminuye en gran medida entre los distintos casos distribuidos. Con una diferencia en los tiempos medios de 11.86 % entre los casos más extremos (1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo).
- Con el método de islas y haciendo uso de SCOOP se observa nuevamente que conforme se va distribuyendo la carga en más workers, el tiempo medio de ejecución se va reduciendo. En este escenario se disminuye el tiempo en un 61.11% entre los casos más extremos.
- Finalmente, comentar que en este escenario la diferencia entre ejecutar el algoritmo de modelo de islas sin paralelización frente al modelo de islas con SCOOP al caso más extremo (3RPi con

4Workers por dispositivo), resulta que paralelizando se emplea un 22.03% más de tiempo. Esto quiere decir que:

- Con respecto al escenario A estamos recortando la diferencia de tiempos entre no paralelizar y paralizar con islas (de un 90% a un 22.03%).
- Si hacemos de manera análoga la comparación entre ejecutar el algoritmo sin paralelización frente al modelo de población total con SCOOP, obtenemos que con SCOOP se tarda un 103.45% más de tiempo medio de ejecución. Comparándolo con el escenario A, se recorta la diferencia de tiempo entre no paralelizar y paralizar (168.42% en el escenario A a 103.45% en el escenario B). Sin embargo, siguen siendo unos valores de tiempo medios de ejecución mucho peores que los análogos con modelo de islas y distribución con SCOOP.

Método	1 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	58.76188	0.4922085	-	-	-	-
Secuencial (Scoop)	-	-	132.96653	0.88222	121.06282	0.49312
Islas (No Scoop)	59.85769	0.7909553	-	-	-	-
Islas (Scoop)	-	-	106.2167	1.03366	82.31595	1.10600

Tabla 7. Tabla de tiempos de ejecución, en segundos, de Escenario B. 1 Dispositivo Raspberry.

Método	2 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	122.12870	2.50138	121.3256	1.91493
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	80.70887	1.80640	73.1985	0.42192

Tabla 8. Tabla de tiempos de ejecución, en segundos, de Escenario B. 2 Dispositivos Raspberry.

Método	3 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	118.70417	1.78045	118.51793	2.20689
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	77.62370	1.66622	72.00535	1.28646

Tabla 9. Tabla de tiempos de ejecución, en segundos, de Escenario B. 3 Dispositivos Raspberry.

A continuación se comentan algunos cambios significativos en los resultados que podemos observar en las tablas 10,11 y 12 correspondientes al escenario C. Recordemos que el escenario C cuenta con un array de solución 40 entradas (ya que el tablero es 60x60) una población total de 300 individuos y se resuelve en 300 generaciones.

- Al igual que observamos en los escenarios A y B en el caso de no usar SCOOP, el tiempo medio de ejecución no varía en gran medida (sólo un 1.71% de incremento) entre usar un modelo de población total o subpoblaciones con migración.
- Nuevamente, en el caso de usar un algoritmo genético de población total paralelizado con SCOOP, el tiempo no disminuye en gran medida entre los distintos casos distribuidos. Con una diferencia en los tiempos medios de 12.44% entre los casos más extremos (1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo).
- Con el método de islas y haciendo uso de SCOOP se observa nuevamente que conforme se va distribuyendo la carga en más workers, el tiempo medio de ejecución se va reduciendo. En este escenario se disminuye el tiempo medio de ejecución en un 50% entre los casos más extremos.
- Es reseñable observar que, por primera vez, en este escenario se emplea menos tiempo paralelizando el modelo de islas con SCOOP (en el caso más extremo) que haciendo uso del algoritmo de modelo de islas sin paralelización, empleando un 2.59% menos de tiempo. Esto quiere decir que:
    - Por primera vez estamos mejorando el tiempo al paralelizar la carga frente a realizarlo secuencialmente.
    - Si hacemos de manera análoga la comparación entre ejecutar el algoritmo sin paralelización frente al modelo de población total con SCOOP, obtenemos que con SCOOP se tarda un 71.79% más de tiempo. Este resultado sigue la tendencia del escenario B de recortar tiempo conforme el escenario tenga mayores dimensiones, pero siguen siendo unos tiempos medios mucho mayores a los no paralelizados.

Método	1 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	117.88503	0.7298994	-	-	-	-
Secuencial (Scoop)	-	-	226.02016	1.21395	204.44894	1.10854
Islas (No Scoop)	119.35116	0.9954069	-	-	-	-
Islas (Scoop)	-	-	174.06782	0.80889	130.86022	2.05607

Tabla 10. Tabla de tiempos de ejecución, en segundos, de Escenario C. 1 Dispositivo Raspberry.

Método	2 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	206.41419	1.473906	202.5788	3.57776
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	130.6839	2.90438	118.02956	0.92446

Tabla 11. Tabla de tiempos de ejecución, en segundos, de Escenario C. 2 Dispositivos Raspberry.

Método	3 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	201.88779	0.88398	202.54092	4.72645
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	124.65739	2.06852	116.40493	1.30038

Tabla 12. Tabla de tiempos de ejecución, en segundos, de Escenario C. 3 Dispositivos Raspberry.

Continuando con las tablas 13,14 y 15 correspondientes al escenario D, que recordemos cuenta con un array de solución 80 entradas (ya que el tablero es 80x80), una población total de 300 individuos y se resuelve en 400 generaciones, se realizan los siguientes comentarios:

- Al igual que observamos en los escenarios A, B y C en el caso de no usar SCOOP, el tiempo medio de ejecución varía mínimamente (tan sólo un 0.58% de incremento) entre usar un modelo de población total o subpoblaciones con migración.
- Nuevamente en el caso de usar un algoritmo genético de población total con SCOOP, el tiempo no disminuye en gran medida entre los distintos casos distribuidos. Con una diferencia en los tiempos medios de 11.31% entre los casos más extremos (1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo).
- Con el método de islas y haciendo uso de SCOOP se observa nuevamente que conforme se va distribuyendo la carga en más workers, el tiempo medio de ejecución se va reduciendo. En este escenario se disminuye el tiempo en un 45.88% entre los casos más extremos.
- En este escenario se observa que se incrementa la diferencia de tiempos entre ejecutar el algoritmo de modelo de islas sin paralelización frente a ejecutar el modelo de islas con SCOOP en el caso más extremo. En este caso, si paralelizamos, se disminuye un 13.71% el tiempo medio de ejecución. Esto quiere decir que:
    - Se sigue mejorando el tiempo al paralelizar la carga frente a realizarlo secuencialmente.
    - Si hacemos de manera análoga la comparación entre ejecutar el algoritmo sin paralelización frente al modelo de población total con SCOOP, obtenemos que paralelizando se emplea un 50.51% más de tiempo. Nuevamente en este escenario se sigue la tendencia de los escenarios anteriores de recortar tiempo conforme el escenario tenga mayores dimensiones, pero de nuevo, siguen siendo unos tiempos mayores a los no paralelizados.

Método	1 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	198.46936	0.65933	-	-	-	-
Secuencial (Scoop)	-	-	336.60598	1.27686	305.08436	1.71812
Islas (No Scoop)	197.86825	1.54194	-	-	-	-
Islas (Scoop)	-	-	248.54659	1.31921	188.01251	1.77213

Tabla 13. Tabla de tiempos de ejecución, en segundos, de Escenario D. 1 Dispositivo Raspberry.

Método	2 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	307.88494	2.45972	305.45378	2.97390
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	190.10360	3.52493	170.13348	2.23116

Tabla 14. Tabla de tiempos de ejecución, en segundos, de Escenario D. 2 Dispositivos Raspberry.

Método	3 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	300.47853	3.65645	298.77947	7.49078
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	182.17689	2.57913	170.50464	2.83619

Tabla 15. Tabla de tiempos de ejecución, en segundos, de Escenario D. 3 Dispositivos Raspberry.

Con respecto a las últimas tablas (16,17 y 18) pertenecientes al escenario E, que recordemos cuenta con un array de solución 100 entradas (ya que el tablero es 100x100), una población total de 300 individuos y se resuelve en 500 generaciones, podemos comentar lo siguiente:

- Al igual que observamos en los anteriores escenarios, en el caso de no usar SCOOP, el tiempo medio de ejecución varía mínimamente (tan sólo un 0.68% de incremento) entre usar un modelo de población total o subpoblaciones con migración.
- En el caso de usar un algoritmo genético de población total con SCOOP, se detecta una diferencia en los tiempos medios de 13.1% entre los casos más extremos (1RPi con 2 Workers frente a 3RPi con 4Workers por dispositivo) que es ligeramente superior a la que se observó en escenarios anteriores.
- Con el método de islas y haciendo uso de SCOOP se observa nuevamente que conforme se va distribuyendo la carga en más workers, el tiempo medio de ejecución se va reduciendo. En este escenario se disminuye el tiempo medio de ejecución en un 33.04% entre los casos más extremos.
- En este escenario se observa que se incrementa la diferencia de tiempos medios de cómputo entre ejecutar el algoritmo de modelo de islas sin paralelización alguna frente al modelo de islas con SCOOP en el caso más extremo. En este escenario, si paralelizamos, se disminuye el tiempo de ejecución un 23.73%. Esto quiere decir que:
    - Se sigue mejorando el tiempo al paralelizar la carga frente a realizarlo secuencialmente.
    - Si hacemos de manera análoga la comparación entre ejecutar el algoritmo sin paralelización frente al modelo de población total con SCOOP, obtenemos que paralelizando se emplea un 35.84% más de tiempo. Nuevamente en este escenario sigue la tendencia de los escenarios anteriores de recortar tiempo conforme el escenario tenga mayores dimensiones, pero siguen siendo unos tiempos mayores a los no paralelizados.

Método	1 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	293.820	1.58351	-	-	-	-
Secuencial (Scoop)	-	-	458.33676	6.53407	414.9073	6.55418
Islas (No Scoop)	295.91976	1.09551	-	-	-	-
Islas (Scoop)	-	-	336.5484	1.15108	250.46509	2.63758

Tabla 16. Tabla de tiempos de ejecución, en segundos, de Escenario E. 1 Dispositivo Raspberry

Método	2 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	422.83656	7.31842	412.2308	6.42537
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	54.13818	2.16556	231.5918	2.24818

Tabla 17. Tabla de tiempos de ejecución, en segundos, de Escenario E. 2 Dispositivos Raspberry.

Método	3 RPi					
	1 Worker/Dispositivo		**2 Workers/Dispositivo**		**4 Workers/Dispositivo**	
	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.	Tiempo medio	Desv. std.
Secuencial	-	-	-	-	-	-
Secuencial (Scoop)	-	-	405.62763	6 .857402	398.2811	7.55943
Islas (No Scoop)	-	-	-	-	-	-
Islas (Scoop)	-	-	241.39071	4.16445	225.6373	4.68001

Tabla 18. Tabla de tiempos de ejecución, en segundos, de Escenario E. 3 Dispositivos Raspberry

En la figura 6-1 podemos observar, de forma visual, curvas que representan la evolución de los tiempos medios de ejecución de los ensayos tanto sin paralelización (modelo de población total sin SCOOP y modelo de islas sin SCOOP) como aquellos paralelizados con 3RPi y 4Worker/Dispositivo (modelo de población total con SCOOP y modelo de islas con SCOOP) a lo largo de los diferentes escenarios planteados. Cada escenario quedaría identificado según las dimensiones del tablero.

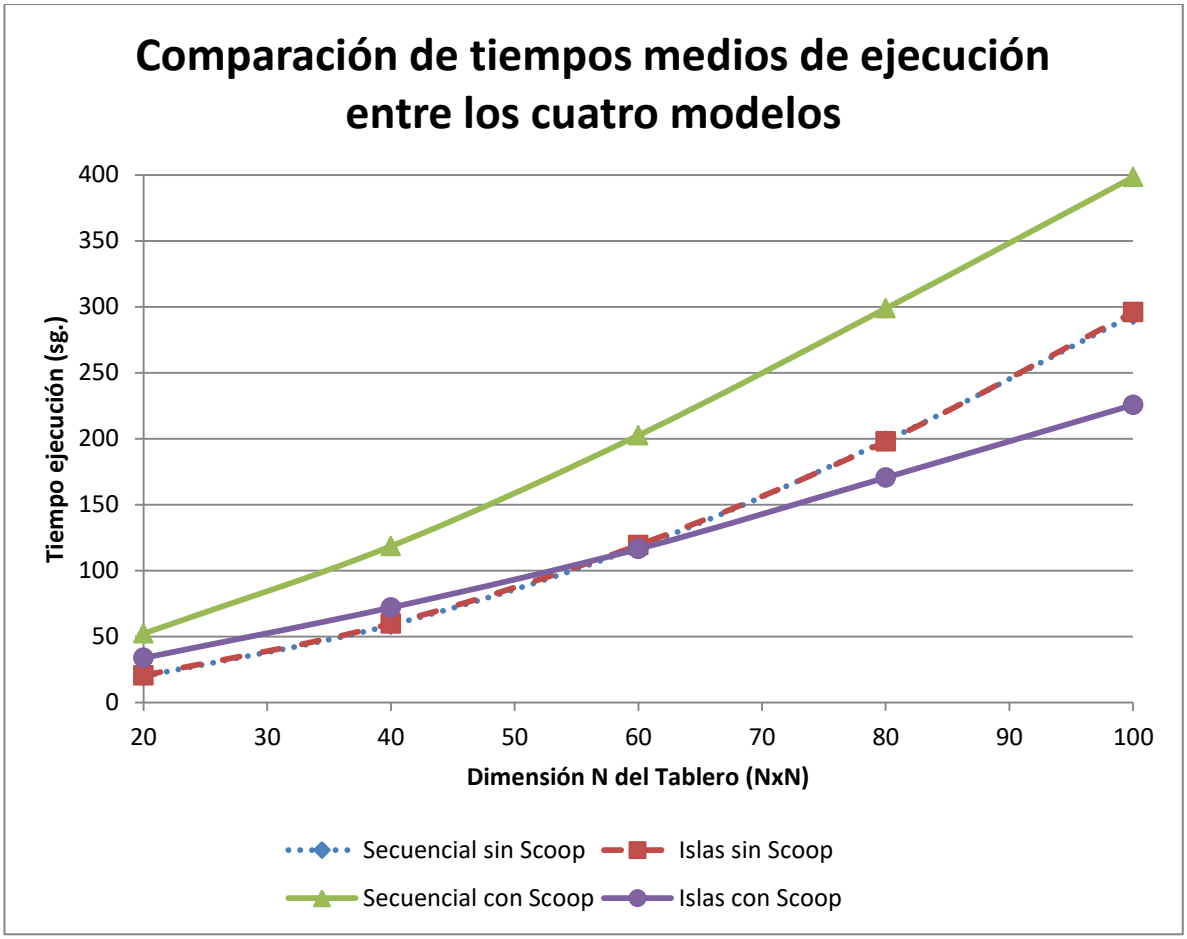

Figura 6-1.Comparación entre tiempos medios de ejecución a lo largo de los cinco escenarios.

Lo más reseñable de esta gráfica es que se puede identificar fácilmente el punto de inflexión que marca la diferencia entre aquellos casos en que se tarde más tiempo (o menos), a pesar de estar usando SCOOP y distribuir la población en subpoblaciones, que directamente no usar SCOOP (distribuir la carga). Dicho punto estaría en torno al escenario C (tablero 60x60, 300 generaciones), donde, como ya se ha comentado, se consigue emplear un 2.59% menos de tiempo medio al paralelizar.

Otra anotación importante sería ver como en el caso de usar un modelo de población total y SCOOP los tiempos medio de ejecución son siempre los más altos. Se podría decir que SCOOP es una herramienta para paralelizar, pero no sabe paralelizar lo que no es paralelizable. Con esto último se quiere decir que al trabajar con una única población y al ser el algoritmo genético secuencial, se pierde más tiempo dialogando entre los workers, que tiempo de cómputo.

Un último comentario sobre la figura 6-1 es que, como ya se ha reseñado en el análisis de las tablas anteriores, en el caso de no usar SCOOP, el tiempo medio de ejecución varía mínimamente entre los casos de usar un modelo de población total o un modelo de subpoblaciones con migración. Este hecho puede observarse en la superposición de las curvas *secuencial sin SCOOP* e *islas sin SCOOP*.

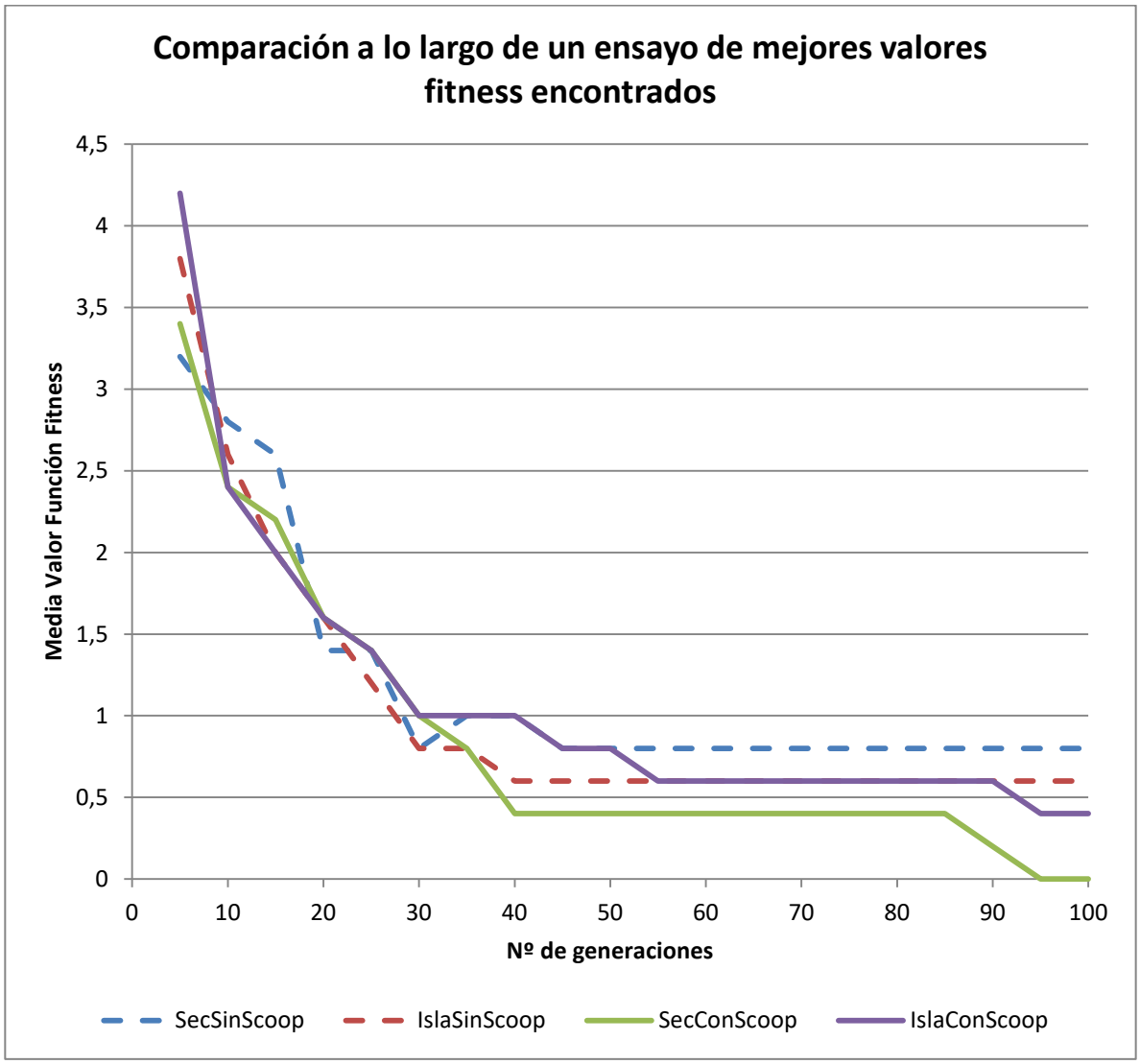

Figura 6-2. Comparación, en el Escenario A, de los mejores valores fitness encontrados.

Tanto en la figura 6-2 como en la figura 6-3 se representa una comparación, entre los distintos modelos, de los valores de fitness encontrados a lo largo de un único escenario. Siendo representado el escenario A en la figura 6-2 y el escenario E en la figura 6-3.

En líneas generales, se observa que el valor de fitness mejora, como es lógico, conforme aumenta el número de generaciones. También puede observarse como en el escenario E, donde el problema tiene una mayor dimensión y por tanto complejidad, que los valores de fitness decrementan exponencialmente de manera más clara que en escenario A. En cualquier caso, no existe diferencia significativa en cuanto a las medias de valores de fitness en los modelos analizados.

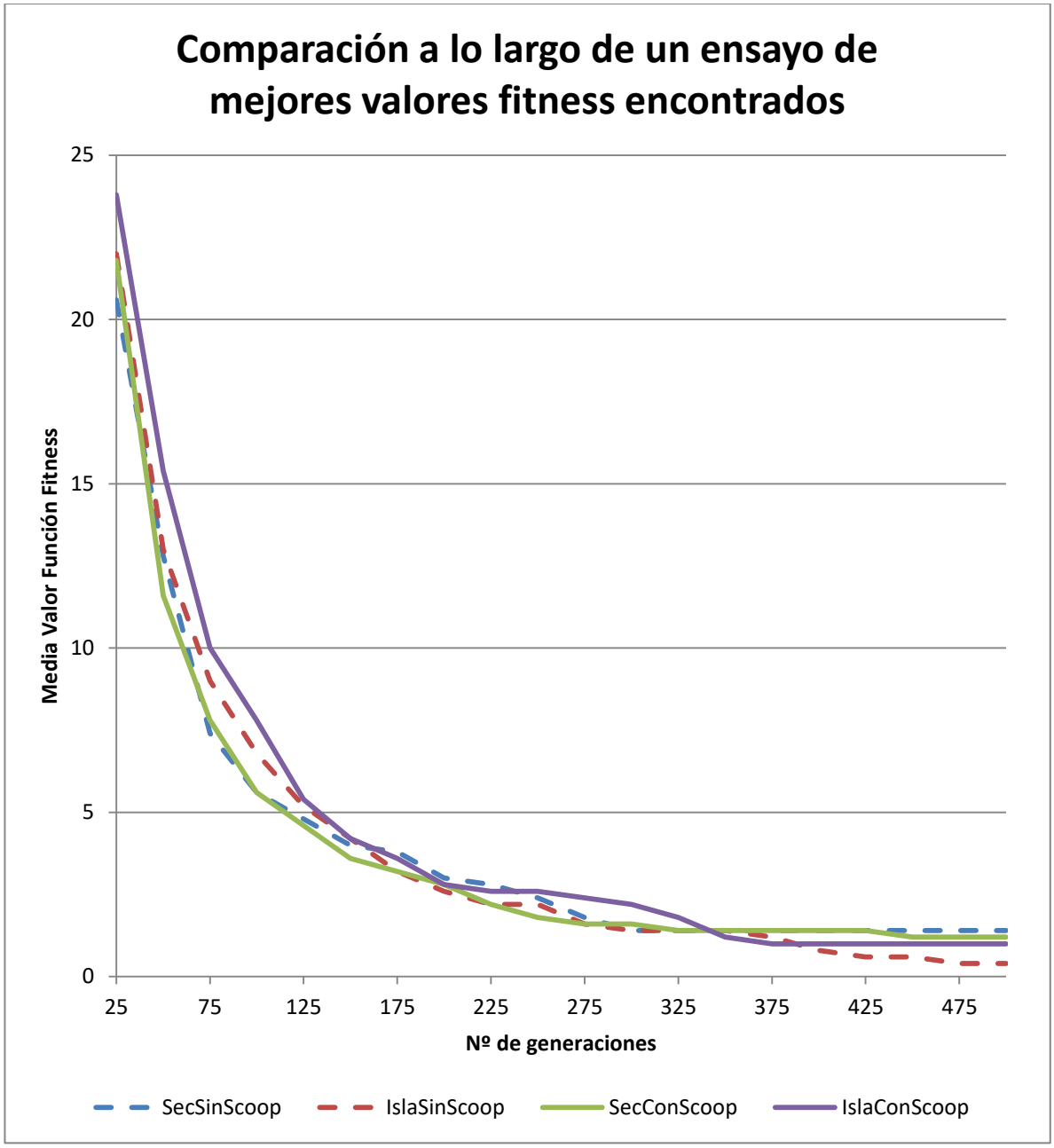

Figura 6-3. Comparación, en el Escenario E, de los mejores valores fitness encontrados.

7 CONCLUSIONES Y FUTUROS TRABAJOS

En este capítulo se presentan algunas conclusiones derivadas del análisis anterior. Finalmente se proponen algunas líneas futuras de trabajo.

7.1 Conclusiones

El uso de algoritmos genéticos para resolver un problema de optimización combinatoria es una técnica cada vez más utilizada debido a que el espacio de soluciones es generalmente demasiado elevado como para ser resuelto por métodos convencionales. A menudo un algoritmo genético aplicado de forma secuencial se topa con la dificultad de que, al ser computacionalmente muy intensivo, en problemas donde se presenta una población de cierta envergadura se necesita de un tiempo de evaluación excesivamente elevado, lo que puede resultar poco adecuado para la resolución de problemas de optimización combinatoria. Una forma de extender la aplicabilidad de la computación evolutiva hacia problemas de mayor complejidad es la inclusión de paralelismo debido a las mejoras que presenta en cuanto a rendimiento.

Esto se debe a dos factores, el primero, una ejecución en paralelo permite un mayor uso de recursos de cómputo como tiempo de procesador o memoria. El segundo factor es que el mismo comportamiento del algoritmo cambia como consecuencia de la estructura de la población. La ejecución en paralelo del algoritmo evolutivo permite una estructura poblacional formada por varios grupos interconectados de individuos.

El resultado final que se ha pretendido encontrar a lo largo de este trabajo, comparando varias estrategias de implementación de algoritmos genéticos, es que se obtenga no solo un tiempo de espera menor para obtener la solución del problema sino que no se vea afectada la efectividad de la búsqueda (valor de fitness). A lo largo de este trabajo se han desarrollado cuatro estrategias de implementación, correspondientes a: modelo secuencial de población única, modelo secuencial de población basado en islas, modelo paralelo de población única y modelo paralelo de población basado en islas. Cada uno de estos modelos ha sido aplicado para resolver el problema de las N reinas y se han considerado cinco escenarios distintos dependiendo de la dimensión del problema y el número de generaciones a considerar.

Como primera conclusión podemos decir que existe un punto de inflexión que marca la diferencia entre aquellos casos en que se tarda más tiempo (o menos), usando un modelo paralelo de población basado en islas, que hacerlo usando un modelo secuencial de población basado en islas. Dicho punto estaría en torno al escenario C, es decir, un escenario dónde el problema a resolver comienza a tener cierta envergadura (dimensión de tablero y número de generaciones). Por tanto, a partir de este punto comienza a ser útil el uso de modelo paralelo de población basado en islas. Por otra parte, la calidad de la solución obtenida por este método sigue siendo del mismo orden que la alcanzada con cualquier otro modelo de los que se han analizado.

Una de las razones por las que podría darse la situación comentada es que, cuando la dimensión del problema no es lo suficientemente grande, el tiempo empleado en el cómputo del valor de fitness no es lo suficientemente elevado en relación al tiempo que se emplea en montar la estructura de paralelización. Dicho punto de inflexión indica el momento en el cual, al crecer la dimensión del problema, el tiempo empleado en el cómputo del valor de fitness excede al tiempo empleado en el montaje de la estructura de paralelización. Por tanto, a partir de este punto es recomendable utilizar un modelo paralelo frente a uno secuencial.

Otra conclusión es que, en el modelo paralelo de población única los tiempos medios de ejecución son siempre los más altos. Esto se debe a que si se trabaja con una única población, la paralelización no es eficaz, pues el algoritmo genético simple sigue una estructura muy secuencial (ya que el operador selección sí debe ser aplicado de forma global a toda la población lo que dificulta la total paralelización del problema) y a pesar de desplegar varios trabajadores se sufren tiempos de espera que penalizan el tiempo de ejecución. Por otro lado, en el modelo paralelo con población basada en islas no se pierde este tiempo de espera porque cada isla es independiente y se optimiza la paralelización. Con ello concluimos que para explotar de forma efectiva la paralelización, a la librería que se usa para paralelizar (SCOOP) se le ha de otorgar un modelo que explote la paralelización (por ejemplo, el modelo de islas).

Finalmente concluimos también que, al no variar en exceso el tiempo medio de ejecución entre los casos de modelo secuencial de población única y modelo secuencial de población basado en islas, si no vamos a usar paralelismo, sería independiente usar un método u otro.

Por lo tanto los resultados que arrojan los experimentos realizados en este trabajo validan el uso de los algoritmos genéticos paralelos para la optimización de problemas de este tipo, en dispositivos de ciertas limitaciones computacionales como puede ser Raspberry Pi.

7.2 Futuras líneas de trabajo

A continuación se presentan diferentes líneas de trabajo futuro relacionadas con el desarrollo de este trabajo:

- Es interesante estudiar la influencia de la heterogeneidad de la población en algoritmos paralelos distribuidos.
- Investigar la implementación de algoritmos que determinen si se es útil seguir buscando o no, evitando estancamientos y creando sistemas distribuidos donde las islas nazcan y mueran, con el objetivo de mejorar la eficacia de los algoritmos y mejorando su adaptación al problema.
- Extender el estudio sobre nuevos dominios de aplicación, como puedan ser la inclusión de la optimización de PathPlanning en una red de robots donde el cómputo necesario para su procesamiento no se centralice sino que se distribuya a través de una red.

REFERENCIAS

Adbelmalik, M., Inza, I., & Larrañaga, P. (2008). *Tema 2. Algoritmos Genéticos / Departamento de Ciencias de la Computación e Inteligencia Artificial Universidad del País Vasco.* Universidad del País Vasco .

Blickle, T., & Thiele, L. (1995). *A Comparison of Selection Schemes Used in Evolutionary Algorithms.*

Campbell, P. J. (1977). Gauss and the eight queens problem: a study in miniature of the propagation of historical error. *Historia Mathematica 4*, 397-404.

Cannon, W. B. (1954). The body psychologic and the body politics. *The Scientific Monthly*, 20-26.

Conor, R., Fitzgerald, J., Medernach, D., & Krawiec, K. (2015). An integrated approach to stage 1 breast cancer detection. *GECCO '15 Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 1199-1206.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* John Murray.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.*

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* Oxford: John Wiley & Sons.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Boston, MA, USA: Addison-Wesley Longman Publishing Co.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Publishing Company.

Gordon, V. S., & Whitley, D. (1993). Serial and Parallel Genetic Algorithms as Function Optimizers. *Proceedings of the Fifth International Conference on Genetic Algorithms*, (págs. 177-183). Illinois, WA, USA.

Gorges-Schleuter, M. (1990). Explicit parallelism of genetic algorithms through population structures. *Parallel Problem Solving from Nature*, 150-159.

Hoffmann, A. (1989). *Arguments on Evolution: A Paleontologist's Perspective.* New York: Oxford University Press.

Holland, J. H. (1975). *Adaptation in Natural and Artificial System.* Cambridge, Massachusetts: The MIT Press.

Khan, S. A., Streater, J., Bathia, T. S., Fiore, S., & Bölöni, L. (2013). Learning social calculus with genetic programing. *FLAIRS 2013 - Proceedings of the 26th International Florida Artificial Intelligence Research Society Conference*, 88-93.

Koza, J. R. (1994). Genetic Programming: On the Programming of Computers by Means of Natural Selection. *Statistics and Computing, Vol 4, Número 2*, 87-112.

Krawiec, K., & Pawlak, M. A. (2015). Genetic Programming with Alternative Search Drivers for Detection of Retinal Blood Vessels. *EvoApplications 2015: Applications of Evolutionary Computation*, 554-566.

Larrañaga, P., Kuijpers, C., Murga, R., & Dizdarevic, S. (1999). Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial intelligence review, 13*, 129-170.

Levine, D. (1993). A Genetic Algorithm For The Set Partitioning Problem. *Proceedings of the Fifth International Conference on Genetic Algorithms*, (págs. 481-487). Illinois, WA, USA.

Lin, S.-C., Punch, W., & E.D.Goodman. (1994). Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach. *Proceedings of the Sixth IEEE Parallel & Distributed Processing*, 28-37.

Lohn, J. D., Hornby, G. S., & Linden, D. S. (2005). An Evolved Antenna for Deployment on Nasa's Space Technology 5 Mission. *Genetic Programming Theory and Practice II*, 301-315.

Lyons, C. (4 de Marzo de 2015). *novadigitalmedia.* Recuperado el 27 de Noviembre de 2018, de http://novadigitalmedia.com/history-raspberry-pi/

Markowska-Kaczmar, U., Kwasnicka, H., & Szczepkowski, M. (2008). Genetic Algorithm as a Tool for Stock Market Modelling. *ICAISC 2008: Artificial Intelligence and Soft Computing*, 450-459.

Mendel, G. (1865). Versuche über Pflanzenhybriden. *Verhandlungen des naturforschenden Vereines in Brünn, Bd. IV*, (págs. 3–47). Brno.

Miller, B. L., & Goldberg, D. E. (1995). Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems, 9*, 193–212.

Nesmachnow, S. (2002). Evolución en el diseño y clasificación. *Conferencia Latinoamericana de Informatica* , (pág. 12). Montevideo, Uruguay.

Nosuna, [. (23 de Septiembre de 2011). *Velneo.* Obtenido de Velneo: https://velneo.es/guido-van-rossum-y-python/

Oliva, R., Ballesta, F., Oriola, J., & Clària, J. (2008). *Genética médica.* Barcelona: Edicions Universitat Barcelona.

Petit, C. (1998). Touched by nature: Putting evolution to work on the assembly line. *U.S. News and World Report, vol.125, no.4*, 43-45.

Red Hat. (24 de Enero de 2010). *OpenSource.* Recuperado el 8 de Enero de 2019, de https://opensource.com/resources/raspberry-pi

Starkweather, T., Whitley, D., & Mathias, K. E. (1991). Optimization Using Distributed Genetic Algorithms. *Parallel Problem Solving from Nature*, 176-185.

Stender, J. (1993). *Parallel Genetic Algorithm: Theory & Applications.*

Tomasini, M. (1999). Parallel and Distributed Evolutionary Algorithms: A Review. *Evolutionary Algorithms in Engineering and Computer Science*, 113-133.

Turing, A. (1948). *Intelligent Machinery.* National Physical Laboratory.

Weismann, A. (1892). *Das Keimplasma: eine Theorie der Vererbung.* Jena: Fischer.

Whitley, D. (1989). The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. *Proceedings of the Third International Conference on Genetic Algorithms*, (págs. 116-121). San Francisco, CA, USA.

Whitley, D., Beveridge, J. R., Guerra, C., & Graves, C. (1998). *Messy Genetic Algorithms for Subset Feature Selection.*

Yang, G., Jeong, Y., Min, K., Lee, J.-w., & Lee, B. (2018). Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair. *Symmetry 10, no. 4*, 92.

APÉNDICE. CÓDIGOS

Código 1. Ensayo_Sec_NoScoop.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
#    Problema de las N Reinas.
#
#    Modelo de poblacion unica.
#
#    El problema consiste en colocar N Reinas en un tablero de dimensiones
#    NxN, de forma que el número de ataques entre reinas sea el menor posible.
#
#    Simplificaciones del problema:
#        > Se considera que solo hay una reina por fila y columna
#        > Por lo tanto los ataques solo pueden ser diagonales
#
#    CodificaciÃ³n del problema:
#        > Los individuos son una lista en la que el indice representa
#          la columna donde se coloca la reina y el contenido es la fila.
#
#    Argumentos de entrada
#        --> Dimension N del tablero
#        --> Numero de generaciones
#        --> Tamaño de la poblacion
#        --> Nombre del fichero en el que se van a registrar los resultados
#        --> Probabilidad, en tanto por 1, de cruce
#        --> Probabilidad, en tanto por 1, de mutacion
######################################################################

import random
import numpy
import sys

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from time import time

#Parametros del problema
NB_QUEENS = int(sys.argv[1])

# FUNCION DE FITNESS
def evalNQueens(individual):
```

```
    """ Funcion de evaluacion del problema. El problema consiste en posiciones
    N reinas en un tablero de ajedrez de dimensiones NxN. La funcion de evaluacion
    calcula el numero de reinas R que hay en cada diagonal. El numero de ataques
    que hay en cada diagonal se puede calcular como R-1.
    """
    size = len(individual)
    # Los ataques solo pueden ser en las diagonales
    diagonal_izquierda_derecha = [0] * (2*size-1)
    diagonal_derecha_izquierda = [0] * (2*size-1)

    # Numero de reinas en cada diagonal
    for i in range(size): # recorremos las columnas
        diagonal_izquierda_derecha[i+individual[i]] += 1 # [columna + fila]
        diagonal_derecha_izquierda[size-1-i+individual[i]] += 1 # [size-1-columna + fila]

    # Numero de ataques en cada diagonal
    suma = 0
    for i in range(2*size-1): # recorremos todas las diagonales
        if diagonal_izquierda_derecha[i] > 1: # hay ataques
            suma += diagonal_izquierda_derecha[i] - 1 # n-1 ataques
        if diagonal_derecha_izquierda[i] > 1:
            suma += diagonal_derecha_izquierda[i] - 1
    return suma,

# DEFINICION DEL PROBLEMA
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# REGISTRO DE FUNCIONES QUE SON NECESARIAS (CAJA DE HERRAMIENTAS)
toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(NB_QUEENS), NB_QUEENS)

# Funciones de inicializacion del individuo y de la poblacion
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.permutation)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

#Funcion de evaluacion
toolbox.register("evaluate", evalNQueens)

# Operadores geneticos
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=2.0/NB_QUEENS)
toolbox.register("select", tools.selTournament, tournsize=3)

def main(num_generaciones, tam_poblacion, prob_cruce, prob_muta):
    pop = toolbox.population(tam_poblacion)
```

```
    hof = tools.HallOfFame(1) # objeto que almacena el mejor individuo

    stats = tools.Statistics(lambda ind: ind.fitness.values) # objeto para calcular estadisticas
    stats.register("Avg", numpy.mean)
    stats.register("Std", numpy.std)
    stats.register("Min", numpy.min)
    stats.register("Max", numpy.max)

    logbook = tools.Logbook()
    pop, logbook = algorithms.eaSimple(pop, toolbox, cxpb=prob_cruce, mutpb=prob_muta,
ngen=num_generaciones, stats=stats,
                        halloffame=hof, verbose=False) # algoritmo genetico como "caja negra"
    return hof, logbook


if __name__ == "__main__":
    num_generaciones = int(sys.argv[2])
    tam_poblacion = int (sys.argv[3])
    nombre_fichero = sys.argv[4]
    prob_cruce = float(sys.argv[5])
    prob_muta = float(sys.argv[6])

    start_time = time()
    best, log = main(num_generaciones, tam_poblacion, prob_cruce, prob_muta)
    tiempo = str(time() - start_time)
    generaciones = list(range(num_generaciones+1))
    fitnesses = log.select("Min")
    with open(nombre_fichero, 'a') as filehandle:
        filehandle.writelines("gen:" )
        filehandle.writelines("%s," % element for element in generaciones)
        filehandle.writelines("\nfit:" )
        filehandle.writelines("%s," % element for element in fitnesses)
        filehandle.writelines("\nElapsed time:")
        filehandle.writelines("%s" % tiempo)
        filehandle.writelines("\n---\n" )
```

Código 2. Ensayo_Sec_Scoop.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
#    Problema de las N Reinas.
#
#    Modelo de poblacion unica y uso de capacidades de SCOOP.
#
#    El problema consiste en colocar N Reinas en un tablero de dimensiones
#    NxN, de forma que el numero de ataques entre reinas sea el menor posible.
#
#    Simplificaciones del problema:
```

```
#        > Se considera que solo hay una reina por fila y columna
#        > Por lo tanto los ataques solo pueden ser diagonales
#
#    CodificaciÃ³n del problema:
#        > Los individuos son una lista en la que el Ãindice representa
#          la columna donde se coloca la reina y el contenido es la fila.
#
#    Argumentos de entrada
#        --> Dimension N del tablero
#        --> NÃºmero de generaciones
#        --> Tamaño de la poblacion
#        --> Nombre del fichero en el que se van a registrar los resultados
#        --> Probabilidad, en tanto por 1, de cruce
#        --> Probabilidad, en tanto por 1, de mutacion
######################################################################

import random
import numpy
import sys

from scoop import futures
from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from time import time

#Parametros del problema
NB_QUEENS = int(sys.argv[1])

# FUNCION DE FITNESS
def evalNQueens(individual):
    """ Funcion de evaluacion del problema. El problema consiste en posiciones
    N reinas en un tablero de ajedrez de dimensiones NxN. La funcion de evaluacion
    calcula el numero de reinas R que hay en cada diagonal. El numero de ataques
    que hay en cada diagonal se puede calcular como R-1.
    """
    size = len(individual)
    # Los ataques solo pueden ser en las diagonales
    diagonal_izquierda_derecha = [0] * (2*size-1)
    diagonal_derecha_izquierda = [0] * (2*size-1)

    # Numero de reinas en cada diagonal
    for i in range(size): # recorremos las columnas
        diagonal_izquierda_derecha[i+individual[i]] += 1 # [columna + fila]
        diagonal_derecha_izquierda[size-1-i+individual[i]] += 1 # [size-1-columna + fila]
```

```
    # Numero de ataques en cada diagonal
    suma = 0
    for i in range(2*size-1): # recorremos todas las diagonales
        if diagonal_izquierda_derecha[i] > 1: # hay ataques
            suma += diagonal_izquierda_derecha[i] - 1 # n-1 ataques
        if diagonal_derecha_izquierda[i] > 1:
            suma += diagonal_derecha_izquierda[i] - 1
    return suma,

# DEFINICION DEL PROBLEMA
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# REGISTRO DE FUNCIONES QUE SON NECESARIAS (CAJA DE HERRAMIENTAS)
toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(NB_QUEENS), NB_QUEENS)

# Funciones de inicializacion del individuo y de la poblacion
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.permutation)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

#Funcion de evaluacion
toolbox.register("evaluate", evalNQueens)

# Operadores geneticos
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=2.0/NB_QUEENS)
toolbox.register("select", tools.selTournament, tournsize=3)

# Operador multiproceso
toolbox.register("map", futures.map)

def main(num_generaciones, tam_poblacion, prob_cruce, prob_muta):
    pop = toolbox.population(tam_poblacion)
    hof = tools.HallOfFame(1) #objeto que almacena el mejor individuo

    stats = tools.Statistics(lambda ind: ind.fitness.values) # objeto para calcular estadisticas
    stats.register("Avg", numpy.mean)
    stats.register("Std", numpy.std)
    stats.register("Min", numpy.min)
    stats.register("Max", numpy.max)

    logbook = tools.Logbook()
    pop, logbook = algorithms.eaSimple(pop, toolbox, cxpb=prob_cruce, mutpb=prob_muta,
ngen=num_generaciones, stats=stats,
                        halloffame=hof, verbose=False) # algoritmo genetico como "caja negra"
```

```
    return hof, logbook

if __name__ == "__main__":
    num_generaciones = int(sys.argv[2])
    tam_poblacion = int (sys.argv[3])
    nombre_fichero = sys.argv[4]
    prob_cruce = float(sys.argv[5])
    prob_muta = float(sys.argv[6])

    start_time = time()
    best, log = main(num_generaciones, tam_poblacion, prob_cruce, prob_muta)
    tiempo = str(time() - start_time)

    generaciones = list(range(num_generaciones+1))
    fitnesses = log.select("Min")
    with open(nombre_fichero, 'a') as filehandle:
        filehandle.writelines("gen:" )
        filehandle.writelines("%s," % element for element in generaciones)
        filehandle.writelines("\nfit:" )
        filehandle.writelines("%s," % element for element in fitnesses)
        filehandle.writelines("\nElapsed time:")
        filehandle.writelines("%s" % tiempo)
        filehandle.writelines("\n---\n" )
```

Código 3. Ensayo_Isla_NoScoop.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
#    Problema de las N Reinas.
#
#    Modelo de poblacion basada en islas
#
#    El problema consiste en colocar N Reinas en un tablero de dimensiones
#    NxN, de forma que el numero de ataques entre reinas sea el menor posible.
#
#    Simplificaciones del problema:
#        > Se considera que solo hay una reina por fila y columna
#        > Por lo tanto los ataques solo pueden ser diagonales
#
#    CodificaciÃ³n del problema:
#        > Los individuos son una lista en la que el Ãindice representa
#          la columna donde se coloca la reina y el contenido es la fila.
#
#    Argumentos de entrada
#        --> Dimension N del tablero
#        --> NÃºmero de generaciones
```

```
#        --> Tamaño de la poblacion de cada isla
#        --> Nombre del fichero en el que se van a registrar los resultados
#        --> Numero de islas
#        --> Frecuencia, en generaciones, de migracion entre islas
#        --> Numero de individuos, por islas, que van a migrar
#        --> Probabilidad, en tanto por 1, de cruce
#        --> Probabilidad, en tanto por 1, de mutacion
######################################################################

import random
import numpy
import sys

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from time import time


#Parametros del problema
NB_QUEENS = int(sys.argv[1])


# FUNCION DE FITNESS
def evalNQueens(individual):
    """ Funcion de evaluacion del problema. El problema consiste en posiciones
    N reinas en un tablero de ajedrez de dimensiones NxN. La funcion de evaluacion
    calcula el numero de reinas R que hay en cada diagonal. El numero de ataques
    que hay en cada diagonal se puede calcular como R-1.
    """
    size = len(individual)
    # Los ataques solo pueden ser en las diagonales
    diagonal_izquierda_derecha = [0] * (2*size-1)
    diagonal_derecha_izquierda = [0] * (2*size-1)

    # Numero de reinas en cada diagonal
    for i in range(size): # recorremos las columnas
        diagonal_izquierda_derecha[i+individual[i]] += 1 # [columna + fila]
        diagonal_derecha_izquierda[size-1-i+individual[i]] += 1 # [size-1-columna + fila]

    # Numero de ataques en cada diagonal
    suma = 0
    for i in range(2*size-1): # recorremos todas las diagonales
        if diagonal_izquierda_derecha[i] > 1: # hay ataques
            suma += diagonal_izquierda_derecha[i] - 1 # n-1 ataques
        if diagonal_derecha_izquierda[i] > 1:
```

```
            suma += diagonal_derecha_izquierda[i] - 1
    return suma,

# DEFINICION DEL PROBLEMA
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# REGISTRO DE FUNCIONES QUE SON NECESARIAS -- CAJA DE HERRAMIENTAS
toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(NB_QUEENS), NB_QUEENS)

# Funciones de inicializacion del individuo y de la poblacion
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.permutation)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Funcion de evaluacion
toolbox.register("evaluate", evalNQueens)

# Operadores geneticos
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=2.0/NB_QUEENS)
toolbox.register("select", tools.selTournament, tournsize=3)

def main(num_generaciones, tam_poblacion, num_islas, frec_mig, num_migrant,
         prob_cruce, prob_muta):
    islands = [toolbox.population(tam_poblacion) for i in range(num_islas)]
    hof = tools.HallOfFame(1) # objeto que almacena el mejor individuo
    stats = tools.Statistics(lambda ind: ind.fitness.values) # objeto para calcular estadisticas
    stats.register("Avg", numpy.mean)
    stats.register("Std", numpy.std)
    stats.register("Min", numpy.min)
    stats.register("Max", numpy.max)
    logbook = tools.Logbook()
    #A continuacion se registra el algoritmo genetico como una funcion
    toolbox.register("algorithm", algorithms.eaSimple, toolbox=toolbox,
                     cxpb=prob_cruce, mutpb=prob_muta, ngen=frec_mig, halloffame=hof, verbose=False)
    generaciones = [] #Lista que guarda aquellas generaciones que han sido evaluadas
    fitnesses = [] #Lista que almacena valores fitness despuÃ©s de cada migracion
    for i in range(0, num_generaciones, frec_mig):
        results=[]
        #Se ejecuta, por isla, el algoritmo genetico
        for j in range(len(islands)):
            results.append(algorithms.eaSimple(islands[j], toolbox,cxpb=prob_cruce, mutpb=prob_muta,
ngen=frec_mig, stats=stats, halloffame=hof, verbose=False))
        islands = [pop for pop, logbook in results] #comprehensive list en la que, por isla, se
obtiene la poblaciÃ³n y datos asociados
        generaciones.append(i+frec_mig)
```

```
        fitnesses.append(hof[0].fitness.values[0])
        tools.migRing(islands, num_migrant, tools.selBest) #puesta en marcha de la mmigracion
    return hof, logbook, generaciones, fitnesses


if __name__ == "__main__":
    num_generaciones = int(sys.argv[2])
    tam_poblacion = int(sys.argv[3])
    nombre_fichero = sys.argv[4]
    num_islas = int(sys.argv[5])
    frec_mig = int(sys.argv[6])
    num_migrant = int (sys.argv[7])
    prob_cruce = float(sys.argv[8])
    prob_muta = float(sys.argv[9])
    start_time = time()
    best, logbook, generaciones, fitnesses = main(num_generaciones, tam_poblacion, num_islas,
frec_mig, num_migrant,
                prob_cruce, prob_muta)
    tiempo = str(time() - start_time)

    with open(nombre_fichero, 'a') as filehandle:
        filehandle.writelines("gen:" )
        filehandle.writelines("%s," % element for element in generaciones)
        filehandle.writelines("\nfit:" )
        filehandle.writelines("%s," % element for element in fitnesses)
        filehandle.writelines("\nElapsed time:")
        filehandle.writelines("%s" % tiempo)
        filehandle.writelines("\n---\n" )
```

Código 4. Ensayo_Isla_Scoop.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
#    Problema de las N Reinas.
#
#    Modelo de poblacion basada en islas y uso de capacidades de SCOOP.
#
#    El problema consiste en colocar N Reinas en un tablero de dimensiones
#    NxN, de forma que el nÃºmero de ataques entre reinas sea el menor posible.
#
#    Simplificaciones del problema:
#        > Se considera que solo hay una reina por fila y columna
#        > Por lo tanto los ataques solo pueden ser diagonales
#
#    Codificacion del problema:
#        > Los individuos son una lista en la que el indice representa
#          la columna donde se coloca la reina y el contenido es la fila.
#
```

```
#    Argumentos de entrada
#        --> Dimension N del tablero
#        --> Numero de generaciones
#        --> Tamaño de la poblacion de cada isla
#        --> Nombre del fichero en el que se van a registrar los resultados
#        --> Numero de islas
#        --> Frecuencia, en generaciones, de migracion entre islas
#        --> Numero de individuos, por islas, que van a migrar
#        --> Probabilidad, en tanto por 1, de cruce
#        --> Probabilidad, en tanto por 1, de mutacion
######################################################################

import random
import numpy
import sys

from scoop import futures
from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from time import time


#Parametros del problema
NB_QUEENS = int(sys.argv[1])


# FUNCION DE FITNESS
def evalNQueens(individual):
    """ Funcion de evaluacion del problema. El problema consiste en posiciones
    N reinas en un tablero de ajedrez de dimensiones NxN. La funcion de evaluacion
    calcula el numero de reinas R que hay en cada diagonal. El numero de ataques
    que hay en cada diagonal se puede calcular como R-1.
    """
    size = len(individual)
    # Los ataques solo pueden ser en las diagonales
    diagonal_izquierda_derecha = [0] * (2*size-1)
    diagonal_derecha_izquierda = [0] * (2*size-1)

    # Numero de reinas en cada diagonal
    for i in range(size): # recorremos las columnas
        diagonal_izquierda_derecha[i+individual[i]] += 1 # [columna + fila]
        diagonal_derecha_izquierda[size-1-i+individual[i]] += 1 # [size-1-columna + fila]

    # Numero de ataques en cada diagonal
```

```
    suma = 0
    for i in range(2*size-1): # recorremos todas las diagonales
        if diagonal_izquierda_derecha[i] > 1: # hay ataques
            suma += diagonal_izquierda_derecha[i] - 1 # n-1 ataques
        if diagonal_derecha_izquierda[i] > 1:
            suma += diagonal_derecha_izquierda[i] - 1
    return suma,

# DEFINICION DEL PROBLEMA
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# REGISTRO DE FUNCIONES QUE SON NECESARIAS (CAJA DE HERRAMIENTAS)
toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(NB_QUEENS), NB_QUEENS)

# Funciones de inicializacion del individuo y de la poblacion
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.permutation)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Funcion de evaluacion
toolbox.register("evaluate", evalNQueens)

# Operadores geneticos
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=2.0/NB_QUEENS)
toolbox.register("select", tools.selTournament, tournsize=3)

# Operador multiproceso
toolbox.register("map", futures.map)


def main(num_generaciones, tam_poblacion, num_islas, frec_mig, num_migrant,
         prob_cruce, prob_muta):
    islands = [toolbox.population(tam_poblacion) for i in range(num_islas)]
    hof = tools.HallOfFame(1) # objeto que almacena el mejor individuo
    stats = tools.Statistics(lambda ind: ind.fitness.values) # objeto para calcular estadisticas
    stats.register("Avg", numpy.mean)
    stats.register("Std", numpy.std)
    stats.register("Min", numpy.min)
    stats.register("Max", numpy.max)
    logbook = tools.Logbook()

    # Elimina metodos no seleccionables antes de enviar a la toolbox
    toolbox.unregister("permutation")
    toolbox.unregister("individual")
    toolbox.unregister("population")
```

```
    #A continuacion se registra el algoritmo genetico como una funcion
    toolbox.register("algorithm", algorithms.eaSimple, toolbox=toolbox,
                     cxpb=prob_cruce, mutpb=prob_muta, ngen=frec_mig, halloffame=hof, verbose=False)
    generaciones = [] #Lista que guarda aquellas generaciones que han sido evaluadas
    fitnesses = [] #Lista que almacena valores fitness despues de cada migracion
    for i in range(0, num_generaciones, frec_mig):
        results = toolbox.map(toolbox.algorithm, islands, stats=stats)
        islands = [pop for pop, logbook in results] #comprehensive list en la que, por isla, se
obtiene la poblacion y datos asociados
        generaciones.append(i+frec_mig)
        fitnesses.append(hof[0].fitness.values[0])
        tools.migRing(islands, num_migrant, tools.selBest) #puesta en marcha de la migracion
    return hof, logbook, generaciones, fitnesses


if __name__ == "__main__":
    num_generaciones = int(sys.argv[2])
    tam_poblacion = int(sys.argv[3])
    nombre_fichero = sys.argv[4]
    num_islas = int(sys.argv[5])
    frec_mig = int(sys.argv[6])
    num_migrant = int (sys.argv[7])
    prob_cruce = float(sys.argv[8])
    prob_muta = float(sys.argv[9])


    start_time = time()
    best, logbook, generaciones, fitnesses = main(num_generaciones, tam_poblacion, num_islas,
frec_mig, num_migrant,
                prob_cruce, prob_muta)
    tiempo = str(time() - start_time)
    with open(nombre_fichero, 'a') as filehandle:
        filehandle.writelines("gen:" )
        filehandle.writelines("%s," % element for element in generaciones)
        filehandle.writelines("\nfit:" )
        filehandle.writelines("%s," % element for element in fitnesses)
        filehandle.writelines("\nElapsed time:")
        filehandle.writelines("%s" % tiempo)
        filehandle.writelines("\n---\n" )
```

Código 5. HacerDatos.py

```
#
#    Sencillo script que dado un archivo que se le entrega por parametro de entrada
#    extrae los resultados de los cinco primeros ensayos y calcula la media de los valores
#    de fitness, valor medio de tiempo de ejecución y varianza
```

```
#    Argumentos de entrada
#        --> Fichero de entrada
#        --> Fichero de salida
#########################################################################

from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt
import sys

c1valor_gen = []
c1valor_fit  = []
c2valor_gen = []
c2valor_fit  = []
c3valor_gen = []
c3valor_fit  = []
c4valor_gen = []
c4valor_fit  = []
c5valor_gen = []
c5valor_fit  = []

valoresTiempo = []
valorMedioFitness = []

obtener = sys.argv[1]
guardar = sys.argv[2]

f = open(obtener)
#Curva1
for line in f:
    if line.startswith("gen"):
        name, val = line.split(":")
        gen, retorno = val.split(",\n")
        c1valor_gen = gen.split(',')
    elif line.startswith("fit"):
        name, val = line.split(":")
        fit, retorno = val.split(",\n")
        c1valor_fit = fit.split(',')
    elif line.startswith("Elapsed time"):
        name, val = line.split(":")
        tiempo, retorno = val.split("\n")
        valoresTiempo.append(float(tiempo))
    elif line.startswith("---"):
        break

#Curva2
for line in f:
```

```
    if line.startswith("gen"):
        name, val = line.split(":")
        gen, retorno = val.split(",\n")
        c2valor_gen = gen.split(',')
    elif line.startswith("fit"):
        name, val = line.split(":")
        fit, retorno = val.split(",\n")
        c2valor_fit = fit.split(',')
    elif line.startswith("Elapsed time"):
        name, val = line.split(":")
        tiempo, retorno = val.split("\n")
        valoresTiempo.append(float(tiempo))
    elif line.startswith("---"):
        break

#Curva3
for line in f:
    if line.startswith("gen"):
        name, val = line.split(":")
        gen, retorno = val.split(",\n")
        c3valor_gen = gen.split(',')
    elif line.startswith("fit"):
        name, val = line.split(":")
        fit, retorno = val.split(",\n")
        c3valor_fit = fit.split(',')
    elif line.startswith("Elapsed time"):
        name, val = line.split(":")
        tiempo, retorno = val.split("\n")
        valoresTiempo.append(float(tiempo))
    elif line.startswith("---"):
        break

#Curva4
for line in f:
    if line.startswith("gen"):
        name, val = line.split(":")
        gen, retorno = val.split(",\n")
        c4valor_gen = gen.split(',')
    elif line.startswith("fit"):
        name, val = line.split(":")
        fit, retorno = val.split(",\n")
        c4valor_fit = fit.split(',')
    elif line.startswith("Elapsed time"):
        name, val = line.split(":")
        tiempo, retorno = val.split("\n")
        valoresTiempo.append(float(tiempo))
```

```
    elif line.startswith("---"):
        break

#Curva5
for line in f:
    if line.startswith("gen"):
        name, val = line.split(":")
        gen, retorno = val.split(",\n")
        c5valor_gen = gen.split(',')
    elif line.startswith("fit"):
        name, val = line.split(":")
        fit, retorno = val.split(",\n")
        c5valor_fit = fit.split(',')
    elif line.startswith("Elapsed time"):
        name, val = line.split(":")
        tiempo, retorno = val.split("\n")
        valoresTiempo.append(float(tiempo))
    elif line.startswith("---"):
        break
f.close()

for i in range(len(c1valor_fit)):
    aux = []
    aux.append(float(c1valor_fit[i]))
    aux.append(float(c2valor_fit[i]))
    aux.append(float(c3valor_fit[i]))
    aux.append(float(c4valor_fit[i]))
    aux.append(float(c5valor_fit[i]))
    valorMedioFitness.append(np.mean(aux))

with open(guardar, 'a') as filehandle:
    filehandle.writelines("fitnessmedio:" )
    filehandle.writelines("%s," % element for element in valorMedioFitness)
    filehandle.writelines("\ntiempomedio:")
    filehandle.writelines("%s" % np.mean(valoresTiempo))
    filehandle.writelines("\nvarianzatiempo:")
    filehandle.writelines("%s" % np.std(valoresTiempo))
```

Código 6. Script_Isla_NoScoop_A.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Isla_NoScoop.py 20 100 100 Resultados_Isla_NoScoop_A 3 5 15 0.5 0.2
done
```

Código 7. Script_Isla_NoScoop_B.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Isla_NoScoop.py 40 200 100 Resultados_Isla_NoScoop_B 3 10 15 0.5 0.2
done
```

Código 8. Script_Isla_NoScoop_C.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Isla_NoScoop.py 60 300 100 Resultados_Isla_NoScoop_C 3 15 15 0.5 0.2
done
```

Código 9. Script_Isla_NoScoop_D.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Isla_NoScoop.py 80 400 100 Resultados_Isla_NoScoop_D 3 20 15 0.5 0.2
done
```

Código 10. Script_Isla_NoScoop_E.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Isla_NoScoop.py 100 500 100 Resultados_Isla_NoScoop_E 3 25 15 0.5 0.2
done
```

Código 11. Script_Isla_Scoop_A_1rpi.sh

```
for i in 1 2 3 4 5
do
python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Isla_Scoop.py 20 100 100
Resultados_Isla_Scoop_A_1RPI_2Wor 3 5 15 0.5 0.2
python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Isla_Scoop.py 20 100 100
Resultados_Isla_Scoop_A_1RPI_4Wor 3 5 15 0.5 0.2
done
```

Código 12. Script_Isla_Scoop_A_2rpi.sh

```
for i in 1 2 3 4 5
do
python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Isla_Scoop.py 20 100 100
Resultados_Isla_Scoop_A_2RPI_2Wor 3 5 15 0.5 0.2
python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Isla_Scoop.py 20 100 100
```

```
Resultados_Isla_Scoop_A_2RPI_4Wor 3 5 15 0.5 0.2

done
```

Código 13. Script_Isla_Scoop_A_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Isla_Scoop.py 20 100 100
Resultados_Isla_Scoop_A_3RPI_2Wor 3 5 15 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Isla_Scoop.py 20 100 100
Resultados_Isla_Scoop_A_3RPI_4Wor 3 5 15 0.5 0.2

done
```

Código 14. Script_Isla_Scoop_B_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_1RPI_2Wor 3 10 15 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_1RPI_4Wor 3 10 15 0.5 0.2

done
```

Código 15. Script_Isla_Scoop_B_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_2RPI_2Wor 3 10 15 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_2RPI_4Wor 3 10 15 0.5 0.2

done
```

Código 16. Script_Isla_Scoop_B_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_3RPI_2Wor 3 10 15 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Isla_Scoop.py 40 200 100
Resultados_Isla_Scoop_B_3RPI_4Wor 3 10 15 0.5 0.2

done
```

Código 17. Script_Isla_Scoop_C_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Isla_Scoop.py 60 300 100
Resultados_Isla_Scoop_C_1RPI_2Wor 3 15 15 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Isla_Scoop.py 60 300 100
```

```
Resultados_Isla_Scoop_C_1RPI_4Wor 3 15 15 0.5 0.2

done
```

Código 18. Script_Isla_Scoop_C_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Isla_Scoop.py 60 300 100
Resultados_Isla_Scoop_C_2RPI_2Wor 3 15 15 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Isla_Scoop.py 60 300 100
Resultados_Isla_Scoop_C_2RPI_4Wor 3 15 15 0.5 0.2

done
```

Código 19. Script_Isla_Scoop_C_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Isla_Scoop.py 60 300 100
Resultados_Isla_Scoop_C_3RPI_2Wor 3 15 15 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Isla_Scoop.py 60 300 100
Resultados_Isla_Scoop_C_3RPI_4Wor 3 15 15 0.5 0.2

done
```

Código 20. Script_Isla_Scoop_D_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_1RPI_2Wor 3 20 15 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_1RPI_4Wor 3 20 15 0.5 0.2

done
```

Código 21. Script_Isla_Scoop_D_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_2RPI_2Wor 3 20 15 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_2RPI_4Wor 3 20 15 0.5 0.2

done
```

Código 22. Script_Isla_Scoop_D_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_3RPI_2Wor 3 20 15 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Isla_Scoop.py 80 400 100
Resultados_Isla_Scoop_D_3RPI_4Wor 3 20 15 0.5 0.2

done
```

Código 23. Script_Isla_Scoop_E_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_1RPI_2Wor 3 25 15 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_1RPI_4Wor 3 25 15 0.5 0.2

done
```

Código 24. Script_Isla_Scoop_E_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_2RPI_2Wor 3 25 15 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_2RPI_4Wor 3 25 15 0.5 0.2

done
```

Código 25. Script_Isla_Scoop_E_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_3RPI_2Wor 3 25 15 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Isla_Scoop.py 100 500 100
Resultados_Isla_Scoop_E_3RPI_4Wor 3 25 15 0.5 0.2

done
```

Código 26. Script_Sec_NoScoop_A.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Sec_NoScoop.py 20 100 300 Resultados_Sec_NoScoop_A 0.5 0.2
done
```

Código 27. Script_Sec_NoScoop_B.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Sec_NoScoop.py 40 200 300 Resultados_Sec_NoScoop_B 0.5 0.2
done
```

Código 28. Script_Sec_NoScoop_C.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Sec_NoScoop.py 60 300 300 Resultados_Sec_NoScoop_C 0.5 0.2
done
```

Código 27. Script_Sec_NoScoop_D.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Sec_NoScoop.py 80 400 300 Resultados_Sec_NoScoop_D 0.5 0.2
done
```

Código 28. Script_Sec_NoScoop_E.sh

```
for i in 1 2 3 4 5
do
python Ensayo_Sec_NoScoop.py 100 500 300 Resultados_Sec_NoScoop_E 0.5 0.2
done
```

Código 29. Script_Sec_Scoop_A_1rpi.sh

```
for i in 1 2 3 4 5
do
python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_1RPI_2Wor 0.5 0.2
python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_1RPI_4Wor 0.5 0.2
done
```

Código 30. Script_Sec_Scoop_A_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_2RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_2RPI_4Wor 0.5 0.2

done
```

Código 30. Script_Sec_Scoop_A_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_3RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Sec_Scoop.py 20 100 300
Resultados_Sec_Scoop_A_3RPI_4Wor 0.5 0.2

done
```

Código 31. Script_Sec_Scoop_B_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_1RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_1RPI_4Wor 0.5 0.2

done
```

Código 32. Script_Sec_Scoop_B_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_2RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_2RPI_4Wor 0.5 0.2

done
```

Código 33. Script_Sec_Scoop_B_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_3RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Sec_Scoop.py 40 200 300
Resultados_Sec_Scoop_B_3RPI_4Wor 0.5 0.2

done
```

Código 34. Script_Sec_Scoop_C_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_1RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_1RPI_4Wor 0.5 0.2

done
```

Código 35. Script_Sec_Scoop_C_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_2RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_2RPI_4Wor 0.5 0.2

done
```

Código 36. Script_Sec_Scoop_C_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_3RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Sec_Scoop.py 60 300 300
Resultados_Sec_Scoop_C_3RPI_4Wor 0.5 0.2

done
```

Código 37. Script_Sec_Scoop_D_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_1RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_1RPI_4Wor 0.5 0.2

done
```

Código 38. Script_Sec_Scoop_D_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_2RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_2RPI_4Wor 0.5 0.2

done
```

Código 39. Script_Sec_Scoop_D_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_3RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Sec_Scoop.py 80 400 300
Resultados_Sec_Scoop_D_3RPI_4Wor 0.5 0.2

done
```

Código 40. Script_Sec_Scoop_E_1rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_1rpi_2worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_1RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_1rpi_4worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_1RPI_4Wor 0.5 0.2

done
```

Código 41. Script_Sec_Scoop_E_2rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_2rpi_2worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_2RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_2rpi_4worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_2RPI_4Wor 0.5 0.2

done
```

Código 42. Script_Sec_Scoop_E_3rpi.sh

```
for i in 1 2 3 4 5

do

python -m scoop --hostfile hostfile_3rpi_2worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_3RPI_2Wor 0.5 0.2

python -m scoop --hostfile hostfile_3rpi_4worker Ensayo_Sec_Scoop.py 100 500 300
Resultados_Sec_Scoop_E_3RPI_4Wor 0.5 0.2

done
```

Código 43. hostfile_1rpi_2worker

```
192.168.1.49        2
```

Código 44. hostfile_1rpi_4worker

```
192.168.1.49        4
```

Código 45. hostfile_2rpi_2worker

```
192.168.1.49        2
192.168.1.51        2
```

Código 46. hostfile_2rpi_4worker

```
192.168.1.49        4
192.168.1.51        4
```

Código 47. hostfile_3rpi_2worker:

```
192.168.1.49        4
192.168.1.51        4
192.168.1.50        4
```

Código 48. hostfile_3rpi_4worker:

```
192.168.1.49        4
192.168.1.51        4
192.168.1.50        4
```