UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación
Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Desarrollo de un Framework para Algoritmos Genéticos en Spark

Trabajo de fin de Máster

Guillem Pitarch Rodrigo

Tutor. Vicente Javier Julian Inglada
Tutora. Adriana Susana Giret Boggino
Tutor Externo. Víctor Sanchez Anguix

Curso 2018-2019

Periodo de escritura

28. 03. 2019 – 30. 06. 2019

Directores

Prof. Giret Boggino, Adriana Susana

Prof. Sanchez Anguix, Víctor

Prof. Julian Inglada, Vicente Javier

Abstract

El objetivo de este trabajo es el desarrollo de un *framework* para Python 3.X capaz de dar soporte al cálculo de algoritmos genéticos haciendo uso de las herramientas de paralelización ofrecidas por Apache Spark, con el fin de probar si dicha herramienta puede dar soporte a estos algoritmos de forma efectiva. A lo largo del documento hablaremos del concepto de computación evolutiva, sus diferentes aproximaciones a la paralelización y acotaremos el concepto de *big data* y sus aplicaciones.

Keywords: Algoritmos genéticos, *big data*, paralelización.

Índice general

1 Introducción

Este capítulo constituye la introducción al proyecto. Consta de cuatro apartados que comienzan por un breve resumen del problema abordado, en el que mencionamos algunos conceptos que serán desarrollados en puntos sucesivos, como los algoritmos genéticos.

Continuaremos declarando una pregunta de investigación y, en el siguiente apartado, hablaremos del método escogido para darle respuesta. Concluiremos con una descripción ordenada del resto de capítulos de este trabajo, donde quedarán resumidos su contenido y finalidad.

1.1. Problema planteado

Los algoritmos genéticos conforman una metaheurística bioinspirada que puede ser utilizada para resolver problemas de optimización y búsqueda [1]. Estos algoritmos forman parte de la rama de la computación conocida como "computación evolutiva", y permiten encontrar soluciones válidas a problemas cuyo tiempo de cálculo sería inasumible en otras circunstancias.

Su idea está basada en la teoría de la evolución, propuesta por el naturista francés Charles Darwin en 1859 en su conocido libro "El Origen de las Especies", según la cual todas las especies del planeta están sometidas a un proceso adaptativo conocido como "selección natural", que las hace adaptarse y evolucionar gracias al éxito o fracaso de múltiples individuos con el pasar de las generaciones.

Estos algoritmos han demostrado ser eficientes en multitud de tareas [2] que van más allá de aplicaciones anecdóticas o teóricas, ofreciendo sobre todo una herramienta de optimización que puede ayudar a mejorar modelos tanto mecánicos [3] como matemáticos, ya que otras técnicas de inteligencia artificial pueden arrojar mejores resultados si son hibridadas con algoritmos genéticos [4].

Lamentablemente los algoritmos genéticos tienen sus trabas; ya que las probabilidades de éxito pueden verse limitadas, aparte de por el modelado del problema, por la potencia de cálculo de la que dispongamos, pues un mejor procesador nos permitirá simular y operar sobre una colonia mayor. El tiempo hasta obtener resultados asciende también, lógicamente, en base a la complejidad del problema, ya que además resolver la función de evaluación que guía el progreso del algoritmo puede ser un problema en sí mismo. Estos factores combinados en el mismo escenario que pueden dar lugar a días de cálculo.

Por este motivo y dada la naturaleza de su funcionamiento, en la que indagaremos más adelante, surge la idea de hacer uso de técnicas de computación distribuida para agilizar el proceso [5]. A lo largo de los años han aparecido nuevas aproximaciones a esta idea tanto a nivel tecnológico, conforme hemos contado con mejores recursos y programas más sofisticados, como a nivel matemático, con modelos de interacción entre colonias distribuidas, que pueden ayudar a explorar mejor el espacio de soluciones de un problema.

1.2. Pregunta de investigación

A raíz de lo expuesto anteriormente nos planteamos si sería eficiente aplicar herramientas de *big data* a la hora de resolver algoritmos genéticos.

Aunque pueda parecer una pregunta sencilla, pues muchas de estas herramientas están pensadas con la computación distribuida en mente volviéndolas idóneas para esta labor, el problema surge en el momento en el que introducimos el concepto de *clustering*. Para poder sacar pleno provecho a la computación distribuida las colonias han de intercambiar elementos, ya que de lo contrario no es tanto que tengamos una población mayor sino múltiples poblaciones pequeñas. Habrá un momento en el que sea necesario comparar resultados e incluso migrar población de una colonia a otra.

Las operaciones de redistribución de datos pueden ser especialmente pesadas pues muchos de estos *frameworks* fueron diseñados con la filosofía de ."escribe una vez, lee muchas", y por ello surge la duda de si el sobrecoste de las mismas llegará a eclipsar la ganancia que obtenemos con la paralelización.

Con el objetivo de intentar llegar a una respuesta positiva decidimos hacer uso de Apache Spark, un conocido *framework* de computación en *cluster* que desde el año 2010 fue liberado como código abierto. Las estructuras de datos de este *framework* fueron diseñadas con el objetivo de mantener los cálculos en memoria, permitiendo realizar acciones y transformaciones sobre ellas en un tiempo muy reducido en comparación con otras herramientas conocidas [6]. Esta decisión se explicará en mayor detalle en el capítulo 4.

1.3. Declaración de objetivos

Para poder probar o refutar nuestra pregunta de investigación comenzamos por diseñar una librería que dará soporte a las operaciones de algoritmos genéticos en un entorno distribuido, usando como herramienta Apache Spark.

Spark ofrece apis en Scala, Java, Python y R. Aunque Java podría ser más adecuado para un proyecto a nivel empresarial nosotros nos decidimos por usar Python, concretamente en su versión 3.5, ya que tiene una comunidad robusta que lo ha dotado de librerías, como pyspark, que ofrecen un nivel de abstracción que facilitara este desarrollo. Por otro lado, dada la naturaleza de este proyecto, el beneficio o pérdida que pueda suponer la elección de un lenguaje respecto a otro no es tan relevante como el propio diseño.

La librería constará de dos módulos principales: uno de ellos gestionará las operaciones que tendrán lugar dentro de cada colonia individual, permitiendo que pasos como la inicialización de la población no tengan que hacerse en el nodo maestro y el otro controlará la interacción con Spark así como los procesos de *clustering* a través de varias nativas del mismo. Este trabajo incluirá también documentación acerca de cómo hacer funcionar esta librería y sus dependencias en un entorno Ubuntu.

Para poner a prueba la librería hemos optado por implementar un problema en el que nos planteamos una aplicación ficticia que permite a sus usuarios participar del envío de paquetes de un lugar a otro de una ciudad. Cada usuario registraría en su perfil los trayectos que suele hacer diariamente, aportando localizaciones y horarios, así como un "margen de tiempo"que podría esperar en cada uno de esos puntos. Con esos datos la aplicación calcularía cómo distribuir el conjunto de paquetes entre sus usuarios de manera que lleguen a su destino, sin que ninguno de ellos tenga que

modificar su ruta original y sin que tengan que cargar con más de un paquete por usuario a lo largo de la jornada.

La simulación de esta aplicación se centrará específicamente en el transporte público de la ciudad de Valencia, ya que ello nos permitirá recrear de forma realista"las rutas que sigue la población a lo largo de una jornada ordinaria. Este trabajo no abordará de ninguna manera hasta qué punto sería viable o práctica una aplicación de esta naturaleza, nos limitaremos a emplear el problema matemático subyacente como ejemplo robusto con el que comparar tiempos y resultados entre los diferentes entornos y recursos con los que realizaremos pruebas.

1.4. Estructura

Los capítulos que siguen a este están estructurados de la siguiente manera:

Comenzaremos por un repaso de la metodología de investigación, que explicará en más detalle cómo se ha obtenido la información para este trabajo.

A continuación, tendremos un capítulo sobre el estado del arte que también abarcará nociones básicas sobre los algoritmos genéticos. Dentro de este punto me he tomado la libertad de incluir una breve introducción al concepto de complejidad computacional ya que, aunque no es en sí mismo estrictamente necesario para entender este trabajo, puede ayudar a comprender el porqué de la existencia y relevancia de este método metaheurístico. En este mismo capítulo también veremos algunas de las diferentes aproximaciones que surgieron con los años en torno al concepto del algoritmo genético distribuido.

El siguiente capítulo hablará del concepto de *big data*, tanto para explicarlo conceptualmente como para desmentir cierta noción generalizada sobre la materia. También aquí veremos algunas de las herramientas más utilizadas por la comunidad y explicaremos el funcionamiento de Spark y por qué fue elegido para este desarrollo.

Terminados ya todos los capítulos conceptuales pasamos a diseño del *framrwork*. En este punto veremos cómo fue diseñado, de qué funciones consta y qué tipo de operaciones y de genéticos son aceptados por el mismo.

Por último, tendremos el capítulo de experimentación, donde explicaremos cómo se abordó el problema del reparto de paquetes a nivel de diseño y cómo este evolucionó a lo largo de varias fases. También quedarán expuestas las métricas obtenidas. Tras esto el trabajo concluye con un capítulo centrado en la interpretación de los resultados y otro que plantea futuros trabajos que pueden derivar de este experimento.

2 Metodología de investigación

Con el objetivo de comprobar si la paralelización con herramientas de *big data* resulta beneficiosa para los algoritmos genéticos comenzamos por un estudio de ambos elementos. Con esa información escogimos un modelo de genético distribuido que empleamos como referente para la implementación de un *framework* capaz de ofrecer soporte a dichos cálculos, tanto con un procesador local como con múltiples núcleos distribuidos.

Este *framework* consta de varios módulos, cuyas descripciones están en el apartado 5. El resultado final quedó almacenado en `https://github.com/Guipirod/gen-fw.git`.

El siguiente paso consistió en modelar un problema difícil de evaluar. Qué constituye un problema "difícil.[es] un tema que abordaremos brevemente más adelante, aunque dicho de forma escueta no está relacionado con volumen de datos a tratar sino con la forma en que el tiempo de cálculo aumenta en relación a dicho volumen.

El problema escogido combina reparto de tareas con búsqueda de caminos; se trata de una aplicación de reparto de paquetes descrita en el apartado 1.3 que aprovecha las rutas realizadas por los transeúntes para entregar una serie de paquetes. Encarar este problema supuso un desafío en sí mismo y también pasó por varios diseños, que quedan explicados en el apartado 6.1.

Con estos objetivos ya cumplidos procedimos a comparar los resultados arrojados por ambos modelos, centrándonos en la evaluación obtenida según el tiempo de cálculo y no el número de iteraciones ejecutadas. Nos centraremos especialmente en la relación entre diferentes tamaños de población en local contra volúmenes similares de población distribuida, a fin de observar si la velocidad obtenida por una población pequeña puede superar la mayor cobertura del modelo distribuido.

Los experimentos fueron realizados en un *cluster* virtualizado, consistente en tres máquinas virtuales que actúan como trabajadores y una máquina virtual que actúa como maestro. Todas las máquinas tienen el sistema operativo Ubuntu 16.04 con 32GB de memoria DDR3 y 8 núcleos de CPU virtuales. El conjunto es ejecutado sobre una máquina física con un procesador de cuatro núcleos Intel Xeon E5-2609, propiedad de la Universidad Politécnica de Valencia.

3 Algoritmos evolutivos

Este capítulo gira en torno al concepto de algoritmo genético, uno de los dos elementos fundamentales de este trabajo.

El primer apartado actúa como una introducción al concepto de complejidad computacional. Los puntos posteriores pueden ser entendidos sin él, pero tener una idea de qué tipos de problemas existen y por qué sus características nos obligan a abordarlos de forma distinta ayuda a entender el porqué de las heurísticas empleadas.

Proseguimos precisamente con un breve recordatorio del concepto de heurística y pasamos a la propia definición de algoritmo genético en el apartado tres. Este punto es el más amplio del capítulo; describiendo desde las bases del funcionamiento de los modelos evolutivos, los operadores empleados más habitualmente hasta diferentes aproximaciones de las versiones distribuidas que pretendemos poner a prueba.

3.1. Una introducción a la complejidad

Me he tomado la libertad de incluir esta mención al concepto de clases de complejidad. Soy consciente de que no es estrictamente necesario entenderlas para comprender el funcionamiento de un algoritmo genético, pero ayuda a entender la razón de ser de esta meta heurística.

La teoría de la complejidad computacional [7] es un campo de la teoría de computación que tiene como objetivo primario estudiar cómo incrementa el coste de los problemas según aumenta su escala, suponiendo siempre que dicha escala sea finita. Este análisis puede resultar de vital importancia a la hora de afrontar un proyecto, ya que aunque un problema sea teóricamente computable eso no implica que necesariamente tenga una solución fácil de alcanzar; por lo general supondremos que un problema es computable cuando puede ser resuelto por un algoritmo equivalente a una máquina de Turing [8], pero la teoría de la complejidad nos permite dar un paso

más y determinar cuáles son los límites prácticos de la capacidad de una computadora.

Todo problema computable puede, por tanto, ser clasificado usando como referencia el incremento del consumo de recursos del algoritmo más eficiente que es capaz de solucionarlo, dando lugar a las categorías que conocemos como clases de complejidad:

- Clase P, según su definición formal contiene los problemas que pueden ser solucionados en tiempo polinómico (de ahí el nombre) por una máquina de Turing determinista, dicho de forma simple: un procesador podría obtener la solución a una instancia de dicho problema en esa escala de tiempo.

- Clase NP, definida como aquella que contiene los problemas que pueden ser solucionados en tiempo polinómico por una máquina de Turing no determinista. Otra manera de entender la clase de complejidad NP es decir que es aquella en la que entran todos los problemas para los cuales, dada una solución cualquiera, verificar dicha solución puede hacerse en tiempo polinómico.

  Nótese que, según las definiciones que hemos planteado, un problema de la clase de complejidad NP puede formar parte también de la clase de complejidad P, pero no necesariamente a la inversa: ya que el que sea sencillo verificar una solución no implica necesariamente que alcanzar esa solución lo sea también. Supongamos por ejemplo que tenemos una lista de números enteros y queremos obtener un subconjunto tal que el sumatorio de todos sus valores sea igual a 0; verificar si la respuesta es correcta puede hacerse linealmente sumando los números de la misma, pero alcanzarla es un problema más complejo.

- Clases NP-Completa y NP-Dura, no entraremos a explicar estas clases en profundidad, ya que hacerlo supondría desviarse demasiado y complicar innecesariamente este apartado introductorio, pero el aspecto que hay que recordar es que en este punto ya nos encontramos con problemas para los cuales verificar una solución supone en sí mismo un problema de tiempo no polinómico.

  Por continuar con el ejemplo anterior: si en lugar de querer cualquier subconjunto de números que sume 0 queremos el mayor subconjunto posible entonces verificar esa solución requiere resolver todo el problema de nuevo, ya que no podemos saber, *a priori*, si lo que nos han dado es efectivamente el más grande hasta que recorramos todo el espacio de soluciones.

Como detalle quisiera añadir que la relación entre las clases de complejidad P y

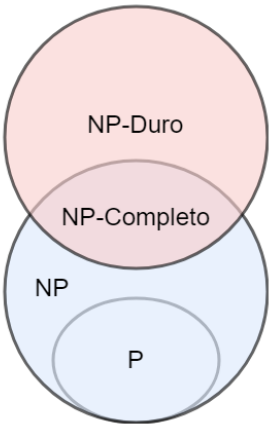

Figura 1: Diagrama de Euler para las clases de complejidad, asumiendo que las clases P y NP están separadas

NP es algo que todavía está siendo debatido, aunque eso no es relevante para este estudio.

3.2. Metaheurística

Con el objetivo de afrontar las dificulades que plantean los problemas de complejidad NP-Duro y NP-Completo recurrimos a métodos heurísticos. Al contrario que los algoritmos exactos, que buscan la solución correcta haciendo todos los pasos secuenciales que el proceso implique, los heurísticos son capaces de devolvernos una solución "suficientemente buena.[en] un tiempo acotado usando como referencia algún tipo de métrica, que les permitirá valorar el rango de discrepancia de cada respuesta obtenida.

La metaheurística en los que se centra este trabajo es llamada algoritmo evolutivo: un modelo bioinspirado que emplea las bases de la teoría de la evolución para refinar resultados. Decimos que se trata de una metaheurística porque no está pensada para resolver un problema específico, sino que se trata de una forma de proceder: una heurística general que puede ser adaptada a múltiples escenarios [9] (de ahí el prefijo "meta", en griego "más allá").

3.3. Algoritmos Genéticos

La teoría de la evolución describe, a *grosso modo*, que todo individuo en la naturaleza compite por los recursos necesarios para su supervivencia ya sea por comida, refugio o incluso la búsqueda de pareja con otros individuos de su especie. Finalmente son aquellos candidatos con mejores cualidades para adaptarse a su entorno los que tendrán más probabilidades de imponerse, sobrevivir y por último encontrar pareja y reproducirse, pasando sus rasgos genéticos a la siguiente generación de la especie [10].

Los algoritmos evolutivos son estrategias de resolución de problemas de búsqueda y optimización inspirados en la teoría de la evolución de las especies. La idea fundamental de estos algoritmos es trabajar con una colección de soluciones candidatas, que irán siendo refinadas a base de múltiples iteraciones haciendo uso de una función de evaluación, que determinará lo acertada que es cada solución al problema planteado. En cada iteración, o generación por emplear un término que se adecue más al modelo biológico, los individuos mejor evaluados tienen mayor probabilidad de reproducirse, propiciando que la composición de la población varíe eventualmente hasta dar lugar a un conjunto más refinado de soluciones [11].

Vale la pena mencionar que los algoritmos evolutivos no pretenden ser una representación fidedigna de la evolución de las especies. Múltiples cualidades del individuo que pueden influir drásticamente en sus posibilidades de reproducción son ignoradas en pro de reducir el tiempo de cálculo, no obstante, algunas cualidades como la edad o la tendencia a la mutación en base al estadio de la evolución sí que han sido estudiadas y pueden arrojar mejores resultados cuando son incorporadas al problema adecuado [12].

Los algoritmos evolutivos permiten abordar problemas complejos de búsqueda y optimización propios de la industria moderna [13] [14], que abarcan conceptos como: planificación de tareas, control de tráfico, optimización de funciones, diseño de componentes mecánicos, búsqueda de caminos etc. Todos estos problemas, generalmente, se encuentran en la clase de dificultad NP-completa. Los algoritmos evolutivos recorren los espacios de soluciones de una forma que no es completamente aleatoria, ya que cuentan con una función de evaluación para orientarse, pero todavía les da la posibilidad de tratar ese sesgo de forma más laxa, permitiendo la exploración de soluciones que en otras circunstancias serían obviadas.

Con estos detalles en mente podríamos decir que los algoritmos evolutivos son un arma de doble filo: por un lado pueden ser implementados con facilitad e integrados en múltiples problemas, ya que su estructura puede ser trasladada o rediseñada para adaptarse a las necesidades del momento, pero por otro lado esa maleabilidad hace que sea necesaria cierta comprensión de las partes que conforman el algoritmo y cómo la elección de dichos componentes influirá en el tiempo y calidad del resultado obtenido [11]. Aunque, como es natural, hay materia escrita al respecto, el conocimiento de la configuración adecuada para cada problema puede venir de detalles sutiles, haciendo que sea menos un proceso plenamente predecible y más cuestión de experiencia.

Existen varios tipos de algoritmo evolutivo, que divergen en la naturaleza de la información codificada pero que no son realmente relevantes para este estudio. Nosotros nos centraremos en el algoritmo evolutivo original: el algoritmo genético.

Estos algoritmos trabajan sobre una lista de soluciones candidatas llamadas individuos, cuya unión recibe el nombre de población. Cada individuo es definido como una serie de valores discretos o continuos, el genotipo, que sirven como cadena de entrada a la función de evaluación: una heurísitca que nos devolverá el grado de optimalidad de cada elemento individualmente. Como primer paso será generada una población pseudo aleatoria, que será posteriormente evaluada y almacenada, conformando la población original. Tras estos pasos tendrán lugar una serie de iteraciones, llamadas generaciones, cuyo objetivo es hacer evolucionar a dicha población aplicando un proceso que imita la selección natural de la siguiente manera:

1. Selección de padres: se elige parejas de individuos para pasar su material genético a la descendencia. Esta selección ha de realizarse considerando de alguna manera el resultado de la función de evaluación sobre los individuos y dando una mayor probabilidad a aquellos con mejor resultado, pero es importante que pese a todo posea cierto grado de aleatoriedad ya que de lo contrario estaríamos privando por completo a individuos con peor resultado de reproducirse, propiciando el quedarnos atascados en óptimos locales.

2. Cruzamiento: los individuos seleccionados para la reproducción son empleados para generar nuevos elementos de la población. Hay varios métodos para realizar este proceso en los que indagaremos más adelante.

3. Mutación: para evitar que todos los resultados alcanzables sean aquellos definidos por las diferentes combinaciones entre los elementos de la primera generación,

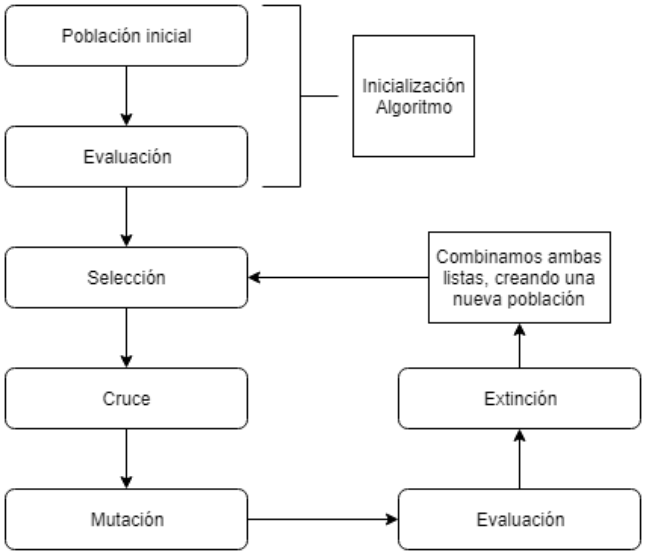

Figura 2: Diagrama de flujo de un Algoritmo Genético

se recrea el proceso biológico de la mutación. Cada elemento de la lista que codifica al individuo tiene asignada una probabilidad de mutar, es decir, de cambia .ªleatoriamente"de valor. Este proceso solo se aplica a los elementos nuevos de la población.

4. Evaluación: aplicamos la función de evaluación sobre los nuevos individuos y almacenamos el resultado.

5. Extinción: parte o la totalidad de la población original es eliminada y reemplazada por los nuevos individuos. Varias aproximaciones a este paso serán tratadas más adelante, pero por lo general si no es eliminada toda la población original entonces se aplicará algún criterio basado en el resultado de la función de evaluación.

Al terminar cada iteración, si el objetivo buscado ha sido alcanzado se da por concluida la búsqueda.

3.3.1. Operaciones de los algoritmos genéticos

Tal y como mencionamos anteriormente el tiempo y la calidad del resultado de un algoritmo genético puede variar en gran medida dependiendo de su configuración. En

esta sección haremos un repaso por las operaciones descritas en el apartado anterior, dando diferentes ejemplos de algunas técnicas empleadas comúnmente, así como sus pros y contras más destacables.

Selección

Es necesario escoger qué individuos van a reproducirse, pasando así sus cromosomas a la siguiente iteración del algoritmo. Este paso trata de garantizar una mayor probabilidad de reproducción a aquellos mejor evaluados, ejerciendo una presión selectiva (una influencia que favorece a ciertos individuos y perjudica a otros) que derivará en una población mejor adaptada. Los ejemplos citados a continuación son algunos de los más empleados a día de hoy, aunque no deben ser considerados monolíticos, sino directrices generales que pueden ser alteradas o combinadas a conveniencia:

- Selección por ruleta: sistema pseudo aleatorio en el que la probabilidad de ser seleccionado de cada cromosoma es directamente proporcional al resultado de su función de evaluación. Esta opción parece inicialmente idónea, ya que no requiere ordenar la población y en teoría los individuos peor evaluados todavía deberían tener probabilidades de reproducción, pero cuando existen grandes diferencias entre los resultados de la función de evaluación aquellos individuos mejor evaluados pueden llevarse fácilmente el 90 % de posibilidades de ser elegidos, estancando así la búsqueda en un óptimo local.

- Selección por rango: surge como una solución al problema inherente de la selección por ruleta. En este caso asignaremos a cada uno de los n individuos un valor desde el n hasta el 1, en orden descendente de mejor a peor evaluación. A partir de este punto se empleará ese valor y no el de la función de evaluación para realizar una selección por ruleta, dando una probabilidad de ser seleccionados a todos los elementos de la población, aunque a costa de incrementar el tiempo de cálculo, ya que hemos reducido la diferencia entre los mejores y los peores individuos.

- Selección por torneo K/L: consiste en seleccionar K elementos aleatoriamente de entre los cuales se reproducirán los L mejores, proceso que se repite hasta generar la nueva población.

- Selección elitista: el problema inverso al mencionado en la selección por ruleta es ese caso en el que perdemos al cromosoma con mejor adaptación, dejando de lado

un valioso material genético. Este método garantiza que los n mejores individuos de cada población pasen sin modificación alguna a la siguiente generación, salvando ese obstáculo. Naturalmente esto puede derivar en una convergencia prematura, ya que facilitamos que un individuo muy bien adaptado se perpetúe e influya más en las generaciones venideras, aunque algunas aproximaciones combaten este fenómeno haciendo que no sobrevivan individuos de la anterior población si se ha generado un cromosoma mejor evaluado.

- Selección por estado estacionario: se conserva a algunos individuos entre generaciones, haciendo que los nacidos en la nueva generación eliminen a los peor adaptados de la población pre existente. Se diferencia del anterior en que los nuevos individuos pueden ser extinguidos por los antiguos. Es rápido pero muy elitista y puede estancar la evolución de la colonia con facilidad.
- Selección generacional: se elimina por completo a la generación anterior, haciendo que cada población esté conformada únicamente por descendientes. Este método es la antítesis de los dos anteriores y suele emplearse en combinación con otros de esta lista.
- Selección jerárquica: a cada individuo de la colonia se le asigna un parámetro que determina cuánto tiempo han permanecido activos, que se actualiza en cada generación del algoritmo. En lugar de contar con una única función de evaluación hay varias que van haciéndose más rigurosas conforme mayor es el tiempo que una muestra ha sobrevivido. Este sistema nos permite emplear operaciones más sencillas con una evaluación rápida, reduciendo el tiempo total de cálculo, y eliminando a los sujetos poco o nada prometedores, mientras aquellos que dan más muestras de resultados positivos reciben un tipo de evaluación más costosa pero detallada.

Cruce

Este es el punto que recrea la reproducción sexual encontrada en la naturaleza: partiendo de la idea de que si cada padre tiene elementos que lo hacen positivo a ojos de la función de evaluación, entonces es posible que su descendencia herede elementos positivos de ambos. En esta versión se genera un número de nuevas muestras a partir de una misma cantidad de elementos de la generación anterior elegidos por el proceso de selección; generalmente dos padres son empleados para generar dos hijos, pero esta cifra puede variar.

Los hijos son creados combinando de alguna manera los valores originales de sus antecesores, siendo relevante la elección de este método, ya que puede arrojar diferentes resultados dependiendo de la naturaleza del problema y de cómo esté codificado el mismo [15].

- Cruce en 1 punto: escogemos aleatoriamente un punto de corte igual para ambos padres, denominado punto de cruce, que separa cada uno en un lado izquierdo y un lado derecho. El lado izquierdo del padre A es enlazado con el lado derecho del padre B mientras que el lado izquierdo de este último es enlazado con el lado izquierdo del padre A. Este método da lugar a dos hijos en un tiempo razonable.

- Cruce en n puntos: similar al corte en 1 punto, pero con n puntos de corte por padre que los dividirán en $n+1$ fragmentos, que se reparten de forma alternada entre la descendencia.

- Cruce uniforme: considerando que ambos individuos tienen la misma longitud, en este método todos los genes tienen un 50 % de probabilidades de pertenecer al hijo 1 o al hijo 2. Una forma de hacerlo es recorrer a los padres simultáneamente y en base a un resultado aleatorio enviar cada gen del puntero a uno de los hijos, aunque este método puede ser algo costoso cuando los individuos tienen un tamaño considerable.

- Cruce por máscara: al inicio de la ejecución es generada una máscara binaria, que posteriormente será empleada para determinar qué gen de qué padre va a cada uno de los hijos. Si bien este método sacrifica la .ªleatoriedad"del proceso, por otro lado facilita una distribución similar a las anteriores sin la necesidad de calcular valores aleatorios constantemente, aligerando de forma considerable este proceso.

Mutación

Para evitar que los resultados alcanzables por el algoritmo estén restringidos a las combinaciones posibles entre valores de los individuos de la población inicial se introduce el concepto de mutación.

La mutación incorpora un elemento de aleatoriedad entre los nuevos individuos de la población, recreando el proceso homónimo encontrado en la reproducción real. Este fenómeno hace que tenga lugar una variación espontánea en el material genético

del individuo que, aunque por lo general no resulta trascendente, puede alterar sus probabilidades de sobrevivir [11].

Tras tener lugar la operación de cruce, los elementos de cada nuevo individuo resultante tienen una probabilidad determinada de cambiar de valor. Este proceso ayuda a evitar la convergencia prematura de la búsqueda [16] y arroja mejores resultados si su probabilidad de ocurrencia va variando conforme va iterando el algoritmo [17].

Resolver individualmente la mutación de cada gen por cada elemento de la nueva población puede acarrear un tiempo de cálculo considerable, por lo que en algunos casos es aconsejable tomar una serie de punteros al azar (el factor de mutación multiplicado por la longitud de la cadena) dentro de cada individuo y alterarlos directamente, o bien hacer uso de máscaras de bits.

3.4. Algoritmos genéticos paralelos y distribuidos

En ocasiones, cuando un problema es demasiado complejo y los individuos de la colonia se vuelven demasiado pesados, puede ser complicado implementar un algoritmo genético eficiente, puesto que la evaluación puede tomar demasiado tiempo o tal vez no se pueda almacenar toda la información manejada en memoria [18].

Incluso si movemos temporalmente parte de los individuos de la colonia por evaluar al disco de la máquina seguiremos incrementando el tiempo final debido a los costes de lectura y escritura. A raíz de este problema surge la idea de aplicar técnicas de paralelización a los algoritmos genéticos, que daría lugar a varias aproximaciones a dicho concepto.

Paralelizar un algoritmo genético no es tarea sencilla, por mucho que la función de evaluación pueda ser aplicada individualmente a cada sujeto de la población, pues el funcionamiento de algunos operadores genéticos precisa interacción. La selección tanto a la hora de reproducirse como, en caso de elitismo, a la hora de realizar la extinción nos forzarán a comparar sujetos.

Los modelos de paralelización que expondremos a continuación no son los únicos, pero sí son empleados con frecuencia e ilustran adecuadamente las diferentes ópti-

cas en que este reto es abordado: dividiendo ciertos cálculos entre nodos esclavos y relegando la selección a un nodo maestro o creando *clusters* de colonias independientes.

3.4.1. Población única con maestro-esclavo

Como su nombre sugiere esta es la típica implementación de paralelización que emplea un maestro que almacena toda la información y distribuye el peso de ciertas operaciones de cálculo entre múltiples trabajadores. En este caso únicamente existe una población de gran tamaño almacenada en el nodo maestro, que es repartida entre varios trabajadores a la hora de evaluar a la población.

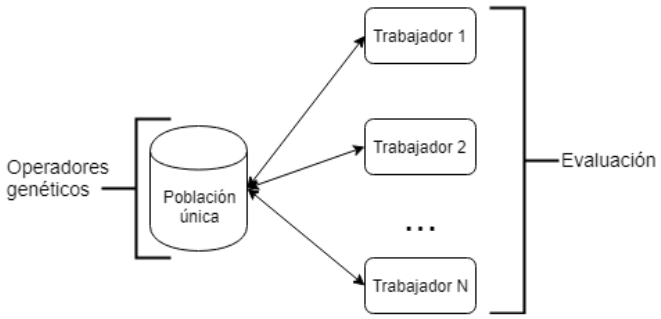

Figura 3: Modelo maestro-esclavo básico

En un principio esta es una paralelización síncrona, de manera que deberíamos esperar el mismo comportamiento que en un algoritmo genético lineal equivalente, a excepción de por velocidad, ya que todos los individuos podrán competir y reproducirse con cualquier otro elemento de la población [19].

La función de evaluación es el cálculo más proclive a ser paralelizado: ya que, salvo en situaciones excepcionales, se tratará del procedimiento más pesado aplicado a elementos individuales, es decir, que no requiere información del resto de la población, como ocurre con la función de selección o el cálculo de los supervivientes. Otras operaciones como la mutación y el cruce entre individuos podrían ser paralelizadas, pero esas operaciones suelen ser tan simples que la carga de tiempo derivada de la comunicación entre múltiples procesadores puede eclipsar la ganancia obtenida [20].

Se ha experimentado con versiones asíncronas de este modelo, en las que el nodo

maestro no espera a todos los esclavos [21]. En este caso el maestro necesitará un *buffer* de entrada para almacenar los envíos y un proceso que distribuya la carga entre los trabajadores de forma no equitativa. Naturalmente este modelo rompe el comportamiento habitual del algoritmo genético, y aunque puede arrojar buenos resultados cabe esperar comportamientos anómalos si la capacidad de cómputo de los procesadores implicados es muy dispar; ya que pueden llegar a convivir individuos de generaciones alejadas.

3.4.2. Población única a grano fino

Los modelos de algoritmo genético a grano fino poseen una única población, pero esta se encuentra dividida entre diferentes "vecindarios", de manera que las interacciones entre individuos quedan limitadas [22].

En esta implementación dividimos los elementos totales entre múltiples procesadores, creando una serie de subpoblaciones capaces de comunicarse las unas con las otras formando un grafo fuertemente conexo, es decir, que dada cualquier pareja de subpoblaciones existe al menos un camino que las conecta.

Cada procesador evalúa y extingue a su población de forma independiente, pero realiza el proceso de selección tomando también elementos de los grupos vecinos. Esta forma de funcionar agiliza el cálculo tanto de la evaluación como de los operadores genéticos, al mismo tiempo que promueve que las mejores soluciones sean distribuidas a lo largo de la malla.

Las diferentes investigaciones realizadas a través de este modelo arrojan luz sobre el rendimiento del mismo [23]. A partir de una malla 2D se observa que el rendimiento del algoritmo disminuye conforme el tamaño de los vecindarios aumenta; de hecho, cuando dichos grupos son lo bastante grandes este modelo paralelo es equivalente a una única gran población lineal.

Otra investigación [24] comparó los efectos del ratio entre el tamaño de la población y de los vecindarios, estableciendo que el tiempo necesario para propagar una solución era exponencialmente inversamente proporcional a dicho valor. Estas dos características convierten el ratio vecindario/población en una decisión de diseño relevante.

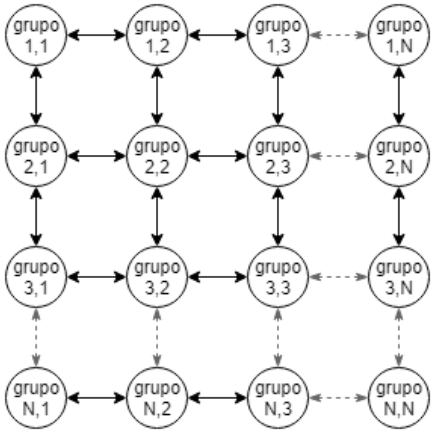

Figura 4: Ejemplo de plano 2D de vecindarios

3.4.3. Modelo en islas

También conocido como modelo distribuido. En este sistema la población es almacenada en colonias diferentes, de manera que no tenemos una única población distribuida sino múltiples poblaciones en equipos distintos que incluso pueden estar configuradas de distinta manera [25].

Esta aproximación ha sido empleada por múltiples investigadores a lo largo de las últimas décadas, así que en pro de la brevedad nos centraremos únicamente en las publicaciones y avances más relevantes para la materia.

La migración de la población es un elemento fundamental de esta implementación, ya que de lo contrario únicamente tendríamos varias colonias trabajando de forma aislada y no estaríamos aprovechando completamente la ventaja de trabajar con un volumen de individuos elevado. Los primeros trabajos concernientes a este tema observaron que la calidad de las soluciones obtenidas con colonias aisladas era inferior a la ofrecida con el mismo número de individuos tratados secuencialmente.

El efecto de este fenómeno sobre el resultado posee cierto paralelismo con la teoría del equilibrio puntuado [26]. Esta teoría sostiene que la evolución no es un proceso constante, sino que las poblaciones pasan largos periodos de equilibrio en los que predomina un genoma determinado hasta que una variación en el entorno o la apari-

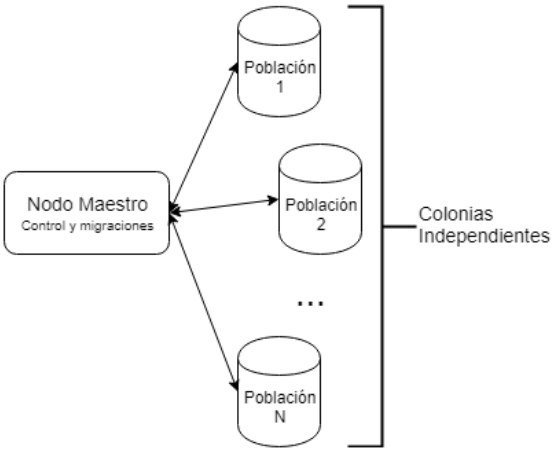

Figura 5: Esquema básico del modelo de islas

ción de una mutación provoca un cambio brusco (el concepto "brusco.[es] relativo, no olvidemos que hablamos de miles de años). Análogamente cuando una población se ha estancado la inserción de elementos de otras colonias ayuda a evitar la convergencia prematura, introduciendo entre sus individuos valores positivos que la población actual no habría alcanzado llegado este punto de estancamiento.

La implementación más común sugiere una estructura en la que un grupo de nodos esclavos ejecutan algoritmos genéticos, enviando cada cierto tiempo a sus mejores individuos a un nodo maestro, que carece de población propia, que tomará los mejores y los distribuirá entre las colonias. La migración de este modelo suele ser síncrona, aunque hay algunos investigadores que han experimentado con modelos asíncronos obteniendo resultados prometedores [27].

La migración queda, por tanto, definida en base a los siguientes factores:

- Tiempo: aquí hablamos no solo de la frecuencia de migración, sino de a partir de qué momento vale la pena el proceso; pues durante las primeras iteraciones de la evolución, cuando la colonia aún no ha comenzado a converger, la introducción de elementos ajenos no será tan significativa como en puntos posteriores.

  Una frecuencia de migración demasiado baja o elevada puede repercutir negativamente en los resultados [28]: si la tasa es demasiado baja el resultado obtenido será similar al alcanzado por la versión secuencial equivalente, pero por otro lado al ser demasiado elevada los costes de tiempo asociados a las

operaciones de transferencia de datos entre las máquinas pueden ser superiores al tiempo reducido, disminuyendo así el rendimiento del algoritmo.

- Tasa de migración: es decir, qué porcentaje de los habitantes de la colonia son enviados y cuántos vienen de fuera en cada proceso. Los primeros modelos enviaban únicamente el sujeto más prometedor de cada grupo, pero enviando más elementos facilitamos que los genomas más prometedores se distribuyan correctamente. Aquí hay que tener cuidado: porque una tasa de migración demasiado elevada también puede desencadenar en elitismo.

  El volumen de datos desplazados también puede, en menor media, influir en el coste de las operaciones de intercambio de información.

- Topología: ha habido múltiples estudios relativos a este factor, ya que de la topología puede depender con qué velocidad y eficiencia son replicadas las soluciones "buenas.enTre las poblaciones [29]. Aquí nos encontramos con un dilema similar al de la frecuencia: si la población es muy conexa las mejores soluciones se distribuirán fácilmente, promoviendo la mezcla de poblaciones, pero incrementarán el tiempo de cálculo con más transferencias de datos, pero si es muy poco conexa favoreceremos la aparición de genomas diferentes.

  Otros estudios [30] han llegado a la conclusión de que una topología fuertemente conexa encuentra una solución global en menos iteraciones.

  Las topologías más empleadas son de la de grafo completo, estructura de anillo y migración por vecindad (similar a la vista en la aproximación por grano fino).

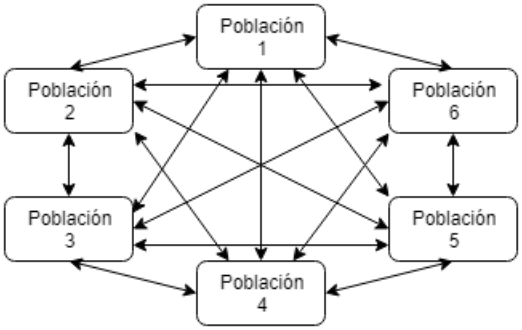

Figura 6: Topología de grafo completo

Existe otra forma de hacer migración: en lugar de intercambiar individuos el nodo maestro recoge toda la población y realiza un proceso de *clustering*; agrupando a los individuos por similitud ya sea por k-vecinos más cercanos, modelos de mixturas

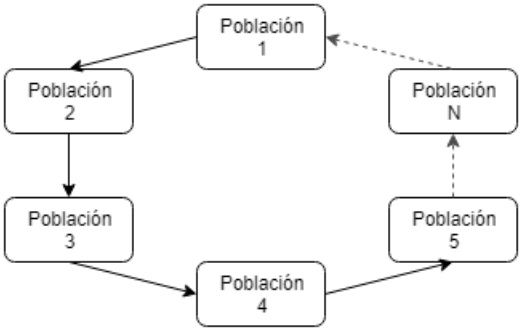

Figura 7: Topología en anillo

gaussianas o cualquier método equivalente.

Si bien este proceso puede ser especialmente pesado, al realizarlo estamos consiguiendo agrupar a los individuos de manera que cada colonia se especialice en un punto prometedor del espacio de soluciones.

Resumen

En este capítulo hemos hablado de la complejidad de los problemas como un concepto relativo a la forma en que escala su consumo de recursos en base al volumen de datos. Los problemas de elevada complejidad pueden ser afrontados de forma efectiva con métodos heurísticos: algoritmos capaces de dar una respuesta "suficientemente buena.[en] un tiempo acotado.

La meta heurística en la que se centra este trabajo es conocida como algoritmo genético: una estrategia de resolución de problemas inspirada en la evolución que recrea la selección natural para refinar un conjunto de muestras. Estos pueden ser adaptados modificando sus operadores, un proceso que requiere cierta comprensión del problema a resolver.

Finalmente hemos visto tres tipos de versiones distribuidas de este algoritmo: el modelo maestro-esclavo, el de población única a grano fino y el modelo de islas. Los dos primeros proponen mantener las muestras unidas, mientras que el tercero aboga por separarlas en colonias independientes pero síncronas, con migraciones eventuales.

4 Big Data

En este capítulo abordaremos el concepto de *big data*, partiendo de una primera sección donde trataremos de acotar qué significa este concepto. Posteriormente veremos el sistema de archivos distribuido Hadoop y el algoritmo Map-Reduce, una publicación de Google, de vasta presencia en este ámbito.

Continuamos con Apache Spark, un *framework* que acelera las operaciones sobre datos distribuidos y de cuáles son sus estructuras de datos, concluyendo en una explicación de cuál de estas tecnologías ha sido escogida para este proyecto y por qué.

4.1. ¿Qué es Big Data?

El *big data*, en ocasiones traducido como "macro datos", es uno de los conceptos que se han puesto de moda más recientemente en el mundo de la informática y que supone la unificación de una multitud de tendencias tecnológicas que fueron madurando desde principios de la década pasada hasta 2012 [31], momento a partir del cual cobró presencia en la sociedad ya fuera a partir de la iniciativa empresarial o el ámbito académico.

Definimos *big data* como la gestión y análisis de enormes volúmenes de información, que no puede ser procesada de manera convencional en un tiempo aceptable. Esta es una definición bastante ambigua, pero luego volveremos a este punto.

Aunque *big data* sea un término relativamente nuevo el hecho de recopilar, procesar y almacenar grandes volúmenes de datos es un problema vigente desde hace mucho tiempo. Su importancia se acentuó conforme un mayor porcentaje de la población mundial comenzó a tener acceso a internet, con sus servicios y redes sociales. En 2012 el presidente ejecutivo de *google*, Eric Schmidt, ya llamaba la atención sobre que desde el principio de la historia de la humanidad hasta el año 2003 se habían generado aproximadamente 5 *exabytes* de datos ($5 * 10^{18}$), pero que cada día ya se generaba

nuevamente esa misma cantidad, un valor que incluso ahora parece duplicarse cada 40 meses [32].

Para hacernos una idea del volumen de datos del que estamos hablando, cierto artículo [33] publicado por la revista *Forbes* en el año 2018 revelaba que cada día eran generados aproximadamente 2'5 quintillones ($2'5 * 10^{30}$) de *bytes*. Entre los ejemplos citados en el artículo encontramos algunos especialmente gráficos:

- 3'5 billones de búsquedas diarias en *google*, unas 40000 por segundo sin tener en consideración otros buscadores populares.
- Más de medio millón de fotos compartidas en *snapchat* por minuto. Con el paso del tiempo la resolución de las fotografías de dispositivos móviles se ha disparado.
- Casi medio millón de mensajes en la popular red social *twitter* son subidos y compartidos por minuto.
- Más de 4 millones de vídeos son vistos y 300 horas de vídeo son subidas cada minuto en *youtube*.

Toda esa información requiere ser procesada en un tiempo mínimo; lo que implica homogeneizar los datos a un formato que pueda ser almacenado e interpretado (ya que un almacenamiento incorrecto puede obstruir el análisis, devaluando la información), comprimir y eliminar redundancias sin comprometer información relevante, llevar a cabo procedimientos de seguridad como la ofuscación de información sensible y, naturalmente, almacenar esos datos, que puede ser otro desafío en si mismo[31].

Asociar *big data* con este tipo de necesidades ha llevado al público general a creer que se trata un término aplicado únicamente a cantidades masivas de información, y que por tanto existiría un baremo según el cual un problema puede ser considerado de esta clase si el volumen a tratar excede una cuota. Esta aproximación es errónea, ya que el tener un gran conjunto de información estructurada y homogénea no implica necesariamente que tratarla entrañe complicación alguna.

Puntualizando la definición ambigua ofrecida anteriormente: una cantidad de datos amerita a ser considerada *big data* cuando es excesiva para que los métodos considerados çomunes"puedan procesarla en un tiempo razonable. Cuando esto ocurre

entran en juego las técnicas de análisis de datos, que pueden ayudarnos a trabajar con grandes volúmenes de información o con pequeños conjuntos que implican problemas complejos (véase la definición de complejidad en el apartado 3.1).

Por poner algunos ejemplos: una búsqueda en la base de datos del censo de una nación cualquiera no es *big data*, pues la información ha sido previamente ordenada e indexada, no obstante convertir millones documentos digitales heterogéneos en esa misma lista ordenada implica decenas de horas de trabajo, aunque sea el mismo volumen de datos. El problema del viajante de comercio para un grafo completo de 20 nodos puede definirse en menos de 1MB de espacio, pero su resolución implica explorar 20! $(2,433 * 10^{18})$ caminos y es por tanto *big data*.

4.2. Apache Hadoop

Apache Hadoop es un *framework* para el almacenamiento y procesado distribuido de datos masivos, de código abierto pero orientado a un hardware comercial. Esta característica lo ha convertido en una herramienta ampliamente utilizada en el mundo de las empresas, ya que facilita la gestión, acceso, integración y procesos básicos de seguridad sobre la información de forma eficiente y con un precio de entrada muy reducido.

Hadoop fue diseñado para funcionar en un *cluster*, generalmente con un único nodo maestro que gestiona y distribuye las tareas del conjunto y múltiples nodos trabajadores (o esclavos) que actúan como almacenamiento distribuido y fuerza de cálculo. Este *framework* presenta dos puntos fuertes adicionales: por un lado la escalabilidad, ya que es capaz de integrar nuevos nodos fácilmente, y por otro la tolerancia a fallos, pues toda la información posee un factor de replicación 3 (es decir, es replicada tres veces automáticamente entre los nodos esclavo) volviendo más improbable la pérdida de datos [34].

Internamente Apache Hadoop puede ser dividido en los módulos que muestra la figura 8:

- Common Utilities: Conformado por todos los *.jar* y librerías necesarias para ejecutar Hadoop. No es relevante por sí mismo, pero ofrece soporte al resto de elementos.

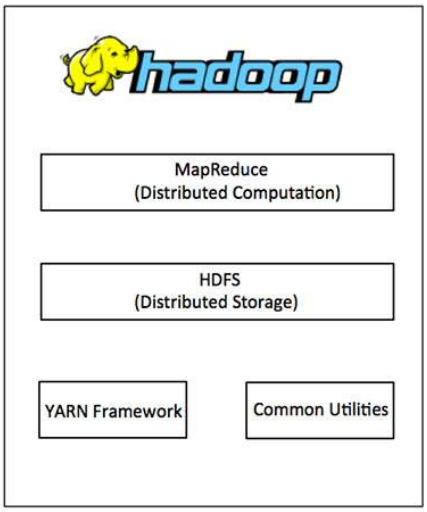

Figura 8: Módulos de Hadoop

- YARN: *Yet Another Resource Negotiator*, se trata del gestor de recursos de Hadoop, que combina un administrador central de recursos, para regular cómo las aplicaciones hacen uso de los recursos del sistema (el nodo maestro), junto con una serie de agentes que monitorizan las operaciones de procesamiento de los nodos individuales.
- HDFS: *Hadoop Distributed File System*, este es el sistema de archivos distribuidos del *framework*, que es instalado en cada uno de los nodos del sistema y empleado por los procesos de Map-Reduce para paralelizar las tareas.
- Map-Reduce: Es una librería que permite hacer uso de este modelo de programación.

4.2.1. Map-Reduce

Map-Reduce fue presentado por dos investigadores de *google* en 2004 como un framework que proporciona un sistema de procesamiento de datos paralelo y distribuido [35]. Su nombre fue inspirado por las dos funciones de programación funcional que integra: *map* y *reduce*. Un esquema con un ejemplo de su funcionamiento puede verse en la figura 9.

- Map: El término "mapear", en su acepción matemática, es un anglicismo que se refiere al proceso de aplicar una operación sobre cada elemento de una estructura

de datos. Esta parte del algoritmo toma una serie de parámetros en forma de pares clave-valor y ejecuta una operación sobre los mismos, generando otra lista de pares clave-valor que luego será separada en diferentes grupos en base a los valores de dichas claves.
Supongamos por ejemplo que almacenamos en una base de datos todas las compras de una tienda *online*. Cada compra posee un identificador propio, el identificador del cliente asociado a la misma y el coste de dicha compra. Una operación *map* aplicada a este conjunto podría generar otro cuyos elementos tengan como clave el identificador del cliente y como contenido una lista con los diferentes gastos que ha efectuado.

- Reduce: Esta función es aplicada en paralelo sobre cada grupo generado por la función *map*. Toma cada clave y el conjunto de valores asociados y los combina mediante una operación de fusión, generando como resultado una colección de datos procesados de menor tamaño.
  Siguiendo el ejemplo anterior, la función *reduce* toma las cadenas resultantes de *map* y suma los valores asociados a cada cliente: hemos obtenido el gasto total para cada uno de ellos.

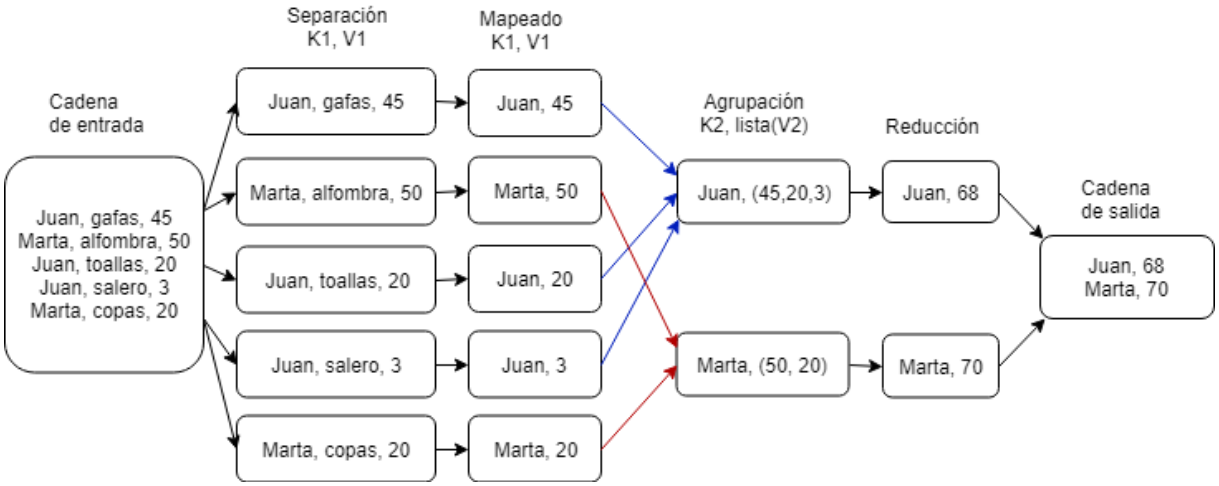

Figura 9: MapReduce

Naturalmente esta es una explicación sencilla del algoritmo Map-Reduce, para algo más detallado me remito a la definición original de la publicación de Jeffrey Dean y Sanjay Ghemawat [35].

Map-Reduce ha sido empleado en múltiples estudios para intentar paralelizar las generaciones de los algoritmos genéticos. Algunos de estos trabajos proponen encapsular cada iteración del algoritmo como una tarea de Map-Reduce [36], adaptando el modelo maestro-esclavo visto en el apartado 3.4.1 al repartir la carga de la función de evaluación.

El modelo de islas descrito en el apartado 3.4.3 entraña una mayor complicación y puede requerir realizar algunas modificaciones sobre el esquema original [37]. Los resultados reportados por algunos estudios [38] revelan que, aunque efectivamente hay una ganancia en la velocidad a la que converge el algoritmo, esta no es directamente proporcional al número de máquinas empleadas.

El porqué de este resultado reside en los tiempos de acceso a disco y latencia de comunicaciones de red asociada a cada operación Map, que pueden llegar a superar la mejora de tiempo obtenida por la división de trabajos. Map-Reduce es útil para paralelizar cuando se cumple alguna de las siguientes condiciones:

- El volumen de datos a almacenar es demasiado elevado para la memoria de una sola máquina. Ya hemos visto que para que un problema sea *big data* este no tiene que ser necesariamente el caso.
- El coste de la memoria y procesadores necesarios para llevar a cabo la tarea en un único equipo es superior al de un conjunto de máquinas de prestaciones individualmente inferiores.
- La ganancia obtenida por la paralelización eclipsa el sobre coste de la comunicación. El tiempo de evaluación aumenta a la par que lo hacen los valores a procesar, por lo que eventualmente se llegará a un punto en que se cumpla esta condición. Esto es más complicado de medir, ya que variará mucho dependiendo de la naturaleza del problema y la estrategia elegida para afrontarlo.

4.3. Apache Spark

Apache Spark es un sofisticado sistema de computación y análisis *open-source* basado en Hadoop que permite dividir trabajos de forma eficiente, enfocado a la velocidad y la facilidad de uso [39]. Nació en el año 2009 a partir de un *paper* de Google gracias a la Universidad de Berkeley, aunque esta eventualmente lo donaría a la .ªpache Software Foundation", que es la actual responsable de su mantenimiento

[40].

A modo anecdótico la figura 10 muestra la línea temporar de la evolución tanto de Hadoop como de Apache Spark.

Al contrario que Hadoop, Spark no es un sistema de archivos distribuidos sino una colección de herramientas y algoritmos pensados para realizar operaciones en *clusters* (aunque puede instalarse en versión *stand alone*). Suele emplearse junto al Yarn de Hadoop por estar integrado en el mismo, pero ofrece soporte para otros *frameworks* como Apache Mesos, siendo capaz de procesar diversas estructuras de datos.

Una de las principales ventajas de Apache Spark es su velocidad: pues trabaja en memoria y tiene una capacidad de procesamiento 100 veces mayor que Hadoop MapReduce. Incluso si Spark se ve obligado a trabajar en disco su velocidad sigue siendo 10 veces mayor [6]. Spark también ofrece APIs para múltiples lenguajes de programación como Scala, R, Python y Java, que facilitan una capa de abstracción beneficiosa para el desarrollador.

Estas virtudes han llevado a Spark a ser adoptado por múltiples empresas, tales como Netflix y Yahoo. En la actualidad posee la mayor comunidad *open-source* de *big data*.

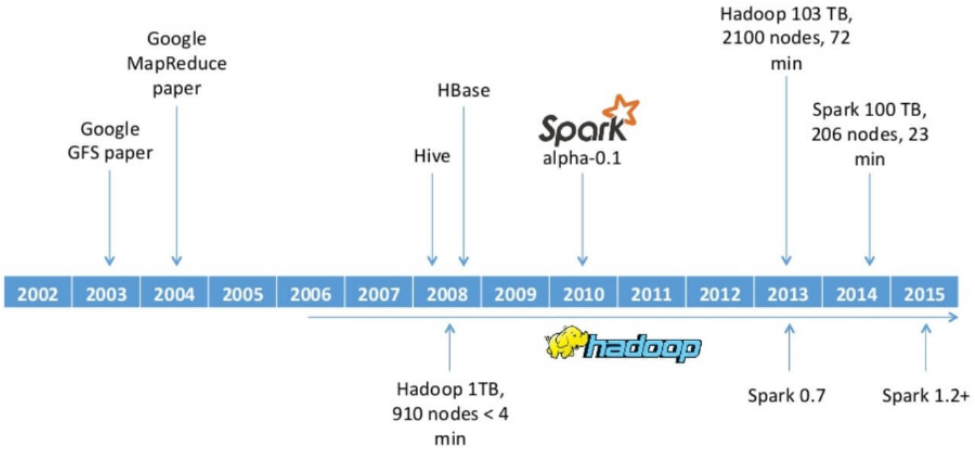

Figura 10: Origen de Spark

La estructura de Spark está compuesta por los componentes de la figura 11:

- Spark Core: conjunto de librerías sobre la que se sustentan el resto de los módulos. Encontramos aquí el planificador de tareas, control de memoria, recuperación tras fallos e interacción con los sistemas de almacenamiento. También reside en este punto la API y la definición de las estructuras de datos a las que Spark tiene acceso, que veremos más adelante.

- Spark SQL: módulo para el procesamiento de información, las librerías que realizan operaciones sobre las estructuras de datos de Spark se encuentran en este punto. Este paquete permite también la interacción a través de una variante de SQL a modo de interfaz.

- Spark Streaming: este módulo está destinado al procesamiento de datos en tiempo real. La información generada por un servicio *web* pasa por operaciones de transformación y es transferida a una RDD.

- MLlib: Spark trae de base una colección de algoritmos de *machine learning* e inteligencia artificial, tales como clasificadores, métodos de regresión o *clustering*. Todos estos métodos están optimizados para Spark y escalan automáticamente en el *cluster*, retirando esa responsabilidad del desarrollador.

- GraphX: Esta librería permite realizar operaciones de lectura y manipulación de grafos. Es importante recalcar que no provee de herramientas de visualización.

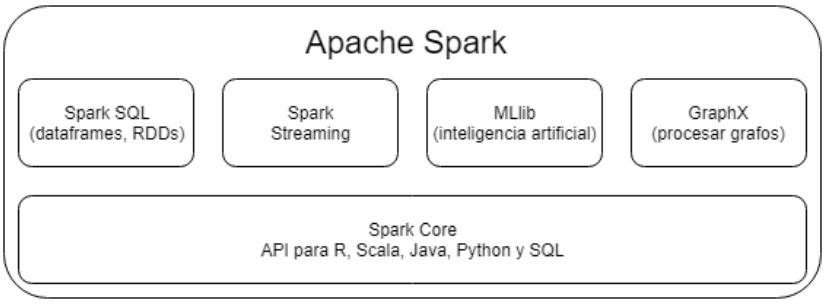

Figura 11: Componentes de Spark

4.3.1. RDD

Su nombre proviene de *Resilient Distributed Dataset* (que podría interpretarse como çonjunto de datos distribuidos resistente.º çonjunto de datos distribuidos elástico") y se trata de la principal estructura de datos de Apache Spark: una abstracción

empleada para representar una colección de objetos ordenados o no.

Un RDD es una colección inmutable de datos, dividida las memorias de nodos del *cluster*, que puede ser modificada en paralelo a través de una serie de operaciones de transformación y acciones (este último punto puede resultar contradictorio, pero se entenderá más adelante). Los RDD almacenan una información que les permite reconstruirse automáticamente en caso de error, siendo especialmente tolerantes a fallos.

Una característica que agiliza las operaciones con RDDs es su evaluación perezosa; cuando se pide una operación de transformación, es decir aplicar un cambio sobre los datos almacenados, esta no es resuelta en el acto sino que queda especificada en una grafo acíclico dirigido como el que representa la figura 12, que define todas las transformaciones que han sufrido los datos.

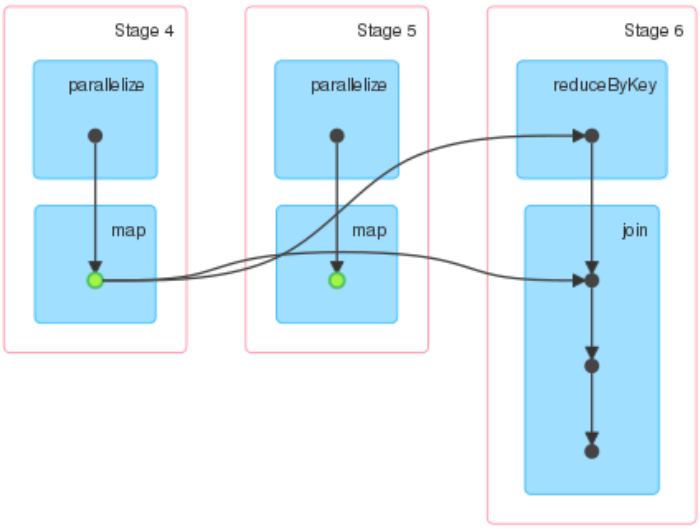

Figura 12: Operaciones acumuladas en una RDD

Las transformaciones únicamente tendrán lugar cuando se solicite una acción sobre la RDD, como por ejemplo recuperar un dato, que no deje más opción a la herramienta que ejecutar las transformaciones. En ese punto Spark sigue el grafo de inicio a fin, ejecutando únicamente las operaciones necesarias hasta devolver el dato, de manera que minimizamos el tiempo de cálculo y obtenemos una estructura de datos especialmente veloz. Cabe destacar que este proceso no altera el contenido de la RDD:

por lo que permite transformaciones, pero sigue siendo inmutable (podemos apreciar ejemplo más gráfico en el esquema mostrado por la figura 13).

La evaluación perezosa es también un arma de doble filo: dado que las transformaciones no se evalúan en el acto es imposible detectar los errores hasta que una acción es ejecutada, lo que puede volver especialmente enrevesada la depuración de programas.

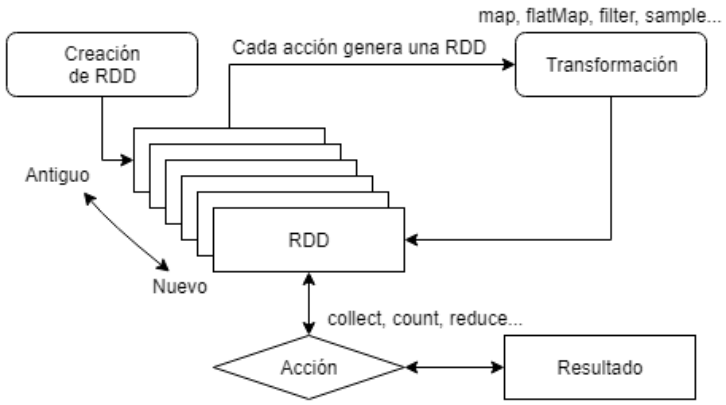

Figura 13: Comportamiento de una RDD

4.3.2. Dataframes y Datasets

Las otras dos estructuras de datos de Apache Spark se llaman Dataframes y Datasets, los primeros están disponibles desde la versión 1.3 del *framework* y los últimos desde la 1.6.

Un Dataframe es una abstracción de información organizada en columnas que acepta información estructurada, similar conceptualmente a las tablas de una base de datos relacional. Se parece a una RDD en que es inmutable, reside en memoria y queda distribuida entre las máquinas del *cluster*, no obstante posee otra cualidad interesante: un mecanismo de optimización de peticiones.

Por otro lado un Dataset es otra estructura de datos de tipado rígido que extiende Dataframe y acepta información ya sea o no estructurada. Posee su propio optimizador, llamado Catalyst Query Optimizer, y en la versión más reciente (2.4.0 en el momento

de la escritura de este trabajo) es considerada más rápida que una RDD o un DataFrame.

4.4. Material empleado

Nuestra intención a la hora de abordar este trabajo siempre fue hacer uso de Apache Spark, pero esta no fue una decisión arbitraria, ya que hay una serie de razones por las que lo preferimos sobre Apache Hadoop además de su popularidad:

- Apache Spark es *per se* más rápido realizando transformaciones sobre sus objetos almacenados que Hadoop MapReduce, además de permitir instalaciones en local descartando la necesidad de un *hardware* específico.
- Posee APIs que ofrecen soporte a múltiples lenguajes de programación, como es el caso de la librería pyspark de Python.
- Spark integra operaciones de *clustering* a través de su librería MLlib, con las que querremos experimentar agrupando la población como mencionamos al final del apartado 3.4.3.
- Aunque seguiremos aumentando el tiempo de ejecución por las operaciones de transferencia de datos, al trabajar en memoria el tiempo de lectura será mucho menor, propiciando así que la paralelización resulte beneficiosa frente al cálculo secuencial.

Finalmente toca decidir entre las estructuras de datos disponibles para Spark. Aunque originalmente puede parecer que emplear la más rápida es la opción adecuada, hay una serie de motivos que nos han llevado a escoger las RDD (que, no olvidemos, siguen siendo 100 veces más veloces que Hadoop Map-Reduce [6]):

- Los Dataframes están limitados a datos estructurados y actúan como la tabla de una base de datos relacional. Estas características los hacen más veloces pero también más rígidos, lo que dificulta la implementación del modelo por islas que veremos en el apartado 5.3.
- Los Datasets aceptan información desestructurada y podrían paralelizar objetos, pero requieren un tipado robusto y por ende no están presentes en algunos lenguajes de programación, entre los que se encuentra Python.

Resumen

En este capítulo hemos definido *big data* como una serie de técnicas de gestión de información que deben ser aplicadas cuando el volumen de datos supera a las herramientas convencionales de un tipo de problema.

Describimos Apache Hadoop como un sistema de archivos distribuido de código abierto que emplea la metodología Map-Reduce. Apache Spark fue desarrollado sobre el anterior y consiste en un *framework* que optimiza el cálculo de operaciones en *clusters* a través de sus estructuras de datos.

Para este trabajo haremos uso de Spark y su estructura de datos primaria: las RDD, una abstracción de datos distribuidos con evaluación perezosa que posibilita un cálculo 100 veces más rápido que el de Hadoop Map-Reduce.

5 Diseño del framework

En este apartado describiremos el *framework* en el que fueron realizadas las pruebas. Comenzaremos justificando el porqué del uso de Python como lenguaje de desarrollo, en el apartado siguiente introduciremos cómo pedir un trabajo a Spark a través de spark-submit, a continuación hablaremos de qué modelos de genéticos fueron adaptados y por último describimos los módulos creados para darles soporte.

5.1. ¿Por qué Python?

Cuando decidimos abordar este proyecto tomamos la decisión de emplear Python 3 como lenguaje de programación, en lugar de otra opción factible y más comercial como lo hubiera sido Java, aunque éramos plenamente conscientes de que es generalmente considerado más lento.

Podemos atribuir el problema de lentitud de Python a varios factores relevantes:

- Python no es un lenguaje compilado, sino interpretado. Esto quiere decir que, en lugar de pasar las instrucciones a un lenguaje de bajo nivel más rápido antes de la ejecución, las líneas de código son procesadas en el acto, acarreando un tiempo de interpretación adicional que de otra manera no sería necesario.
- El "tipado"de Python es débil (también se le llama tipado dinámico). Al declarar una variable no estamos obligados a asignarle un tipo, y este puede incluso variar a lo largo de la ejecución. Esto facilita la legibilidad del código y lo hace más tolerante a cambios, pero puede perjudicar ligeramente al rendimiento.
- El GIL, siglas de *Global Interpreter Lock*, es un mecanismo empleado por el intérprete de Python que lo ayuda a mejorar la velocidad de los programas y facilita la integración de librerías de *c*, pero lo fuerza a ejecutar únicamente un hilo al mismo tiempo aunque nos encontremos en un procesador con varios núcleos.

Aunque en un principio la existencia de GIL hace parecer a Python un lenguaje poco apropiado para este trabajo, hay que tener en cuenta que emplearemos la clase de RDD que ofrece pyspark. Esta clase actúa como una interfaz con Apache Spark, que es el encargado de gestionar esa paralelización, de manera que cuando *mapeamos* una función esta sí es enviada de forma simultánea a los núcleos del *cluster*. El intérprete tendrá que esperar a que todos los nodos devuelvan su respuesta, como ocurre en cualquier sistema síncrono.

Sigue siendo innegable que otros lenguajes son más rápidos si analizamos únicamente su velocidad de procesado, pero también hay factores que han llevado a Python a alcanzar la notoriedad que tiene a día de hoy [41], incluso en proyectos de aprendizaje máquina o minería de datos:

- Su gramática fue diseñada con la legibilidad en mente. Este puede parecer un detalle trivial, pero nada más lejos de la realidad: un código legible es más fácil de entender y mantener.
  La simpleza de este lenguaje permite al programador abstraerse de los pormenores la de implementación y centrarse casi únicamente en la lógica del algoritmo.

- Python es de código abierto y posee una comunidad de usuarios muy amplia. Tanto es así, que en su encuesta anual del 2018 el prestigioso portal *web* Hackerrank se refirió a él en los siguientes términos: *"JavaScript puede ser el lenguaje más demandado por las empresas, pero Python es el que ha conquistado a más desarrolladores a lo largo del mundo."* [42].
  Estos dos factores hacen que encontrar documentación y librerías sea muy sencillo, lo que nos lleva al siguiente punto:

- Una gran comunidad de usuarios conlleva una gran colección de repositorios de código abierto. Sea cual sea la tarea Python tiene una herramienta optimizada:
  - Cálculo científico: numpy (muchas de sus funciones están en *c*) o scipy
  - Procesamiento de imágenes o texto: nltk (*natural language tool kit*), opencv o scikit
  - Procesamiento de audio: librosa
  - Aprendizaje máquina: scikit, tensorflow, pytorch o keras
  - Visualización de información: pandas, matplotlib o seaborn

Python tiene incluso, como hemos mencionado anteriormente, una librería específica para Apache Spark.

En este proyecto la velocidad ha de ser aportada por Apache Spark a través de la paralelización, y la elección de lenguaje no es tan importante como el diseño. Es por estos motivos que, aunque somos conscientes de las limitaciones de Python, lo preferimos por las opciones y facilidad que ofrece al programador.

5.2. Ejecutando en Spark

Tal y como mencionamos en el apartado 4.3 Apache Spark generalmente estará montado sobre un sistema de archivos distribuidos. Para este estudio en concreto contamos con YARN, el gestor de recursos de Hadoop mencionado en el apartado 4.2.

Trabajar en entornos distribuidos supone la necesidad de ofrecer al desarrollador cierto grado de control sobre las dependencias de su programa y sus versiones. Por este motivo YARN ejecuta cada trabajo en un entorno individual para cada usuario, permitiendo que varios proyectos con necesidades distintas puedan convivir en el mismo espacio.

Esto quiere decir que junto a las instrucciones del trabajo deberemos adjuntar los recursos y dependencias necesarios, que serán distribuidos entre todos los nodos por las funciones del HDFS con un factor de replicación 3, es decir, que hasta 3 nodos tendrán copias de cada paquete (ver figura 14). La replicación puede ser alterada manualmente, y no es inusual hacer uso de un valor entre el 10 y el 20.

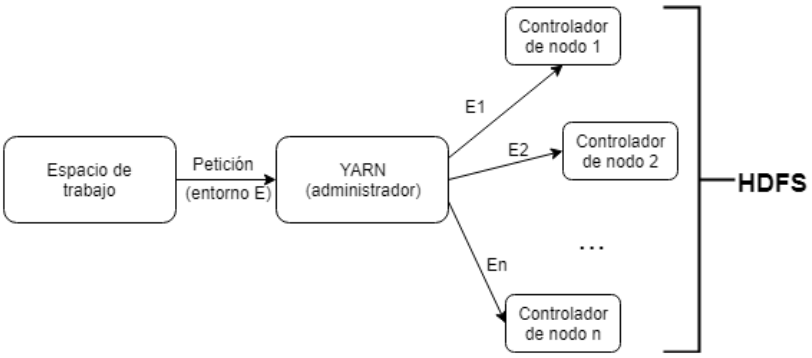

Figura 14: Relación entre HDFS y YARN

5.2.1. Generando un entorno virtual

A continuación veremos un ejemplo simple de cómo preparar un trabajo adjuntando como dependencia la conocida librería de Python NumPy (que es una de las dependencias de este proyecto), empleada entre otras cosas por su soporte a operaciones matriciales y vectores.

```
# -*- coding: utf-8 -*-

import numpy as np
from dependency import foo # 1
from pyspark import SparkContext, SparkConf

if __name__ == '__main__':
    # 2
    conf = SparkConf().setAppName('example').setMaster('yarn')
    sc = SparkContext(conf=conf, pyFiles=['dependency.py'])
    # 3
    vect = np.arange(100)
    rdd = sc.parallelize(vect)
    assert(vect[0] == rdd.first())
    # 4
    vect2 = np.array([foo(v) for v in vect])
    rdd2 = rdd.map(foo)
    assert(vect2[0] == rdd2.first())
```

Listing 1: Código de example.py

Tenemos un *script* sencillo llamado .example.py", cuyo código puede ser leído en el *listing* 1. Suponemos que viene acompañado de otro *script* de nombre "dependency.py", que incluye una función determinista que podemos *mapear* sobre una lista y que a su vez hace uso de funciones de NumPy.

Siguiendo el código mostrado pasamos por los siguientes puntos:

1. Importamos librerías como en cualquier programa de Python. En este caso hemos importado NumPy y dependency.py junto a dos clases de pyspark: SparkContext y SparkConf.

2. Configuramos la prueba. La clase SparkConf (*spark configuration*) nos permite asignar una serie de características a nuestro programa: hemos dado un nombre

a la prueba y, más importante, la función setMaster con *yarn* como parámetro de entrada indica que esto va a ser ejecutado de forma distribuida en el *cluster*.

En la siguiente línea inicializamos el SparkContext (contexto de Spark); esta es una clase que actúa como interfaz entre Apache Spark y nuestro programa. Observemos que, a parte de la clase de configuración definida en el paso anterior, hemos añadido dependency.py a este contexto. Al realizar este paso se generarán copias en los nodos del sistema, de lo contrario el *script* fallaría irremediablemente al intentar hacer uso de dichas funciones.

3. La primera instrucción de este punto hace uso de NumPy para generar un vector de enteros ordenados del 0 al 99, que queda almacenado en la variable vect. El siguiente punto llama al contexto de Spark para generar una RDD (ver apartado 4.3.1), que distribuirá los datos de vect a través de nuestro sistema distribuido. El primer elemento de vect y de la RDD son el mismo.
   Si quisiéramos comparar otro elemento (hemos empleado *first()* por hacer más legible el código) podríamos ejecutar la función *collect()* de la RDD, que devuelve todo su contenido en un iterable.

4. Por último vamos a ejecutar un par de transformaciones. Este código debería funcionar suponiendo que la función foo cumple las condiciones citadas anteriomente. Primero creamos un nuevo vector de NumPy aplicando foo a cada elemento de vect, luego invocamos la función *map* de la RDD pasando como parámetro de entrada la función a *mapear*. Dos cosas a tener en cuenta llegado este punto:

   - Las RDD son inmutables, por lo que hay que asignar este cambio a una variable.
   - Una vez ejecutada la instrucción *map* nada ha sido calculado. Cuando pidamos un valor de la RDD esta calculará las transformaciones necesarias para hacérnoslo llegar (ver apartado 4.3.1).

   Cuando comparamos de nuevo los primeros elementos tanto la RDD de Spark como el objeto de NumPy deberían devolver el mismo valor.

```
#!/bin/bash

$ #1 - creacion
$ virtualenv numpy_env
$ virtualenv --relocatable numpy_env
$ source numpy_env/bin/activate

(numpy_env)$ #2 - instalacion
(numpy_env)$ yum install -y gcc make python-devel
(numpy_env)$ pip install numpy==1.7.2
(numpy_env)$ #3 - empaquetar
(numpy_env)$ zip -r numpy_env.zip numpy_env
(numpy_env)$ source numpy_env/bin/deactivate
```

Listing 2: Creación de un entorno virtual

Para hacer funcionar este *script* generaremos un entorno siguiendo los pasos descritos en el *listing* 2:

1. Creamos un entorno virtual, en este caso lo hemos llamado numpy_env, y lo hacemos rehubiclable. Accedemos al mismo.
2. Procedemos a instalar las dependencias que necesitemos en la versión correcta.
3. Comprimimos el entorno (útil para pasarlo entre equipos, no necesario si vas a ejecutar en la misma máquina) y salimos del mismo

5.2.2. Peticiones a Spark-Submit

```
#!/bin/bash

$ spark-submit --master yarn --conf spark.pyspark.virtualenv.enabled=true\
   --conf spark.pyspark.virtualenv.type=native\
   --conf spark.pyspark.virtualenv.requirements=$ENVIRONMENT_PATH/requirements.txt\
   --conf spark.pyspark.virtualenv.bin.path=/usr/bin/virtualenv\
   --conf spark.pyspark.python=$PYTHON_PATH example.py
```

Listing 3: Petición al *cluster* por Spark-Submit

En el *listing* 3 podemos ver una petición del *script* diseñado en el apartado anterior. Las variables ENVIRONMENT_PATH y PYTHON_PATH son, respectivamente,

las rutas al directorio del entorno y a la versión del ejecutable de Python que va a ser empleado, que idealmente ha de ser la que se encuentra en el entorno (ya que les llegará junto al mismo), aunque no es estrictamente necesario si todos los trabajadores poseen la misma versión.

En caso de estar experimentando con pyspark aconsejamos encarecidamente o bien ajustarlo para reducir su verbosidad o asegurarnos de guardar los *logs* en un archivo accesible, ya los constantes mensajes mostrados por YARN pueden dificultar la depuración de los programas.

5.3. Modelos adaptados

Cuando comenzamos a diseñar el *framework* de las pruebas pensamos en adaptar el modelo de algoritmo genético en islas, del que hablamos en el apartado 3.4.3. Esta decisión fue motivada porque se trata de la aproximación que más se diferencia del sistema original, y buscábamos experimentar con el concepto de *clustering* y las migraciones para ver cómo afectaban al rendimiento.

En el apartado 4.3.2 mencionamos que estábamos algo limitados a la hora de escoger con qué estructuras podíamos contar para esta implementación, así que nos ceñimos a RDD que, aunque no son tan rápidas como un DataFrame, nos ofrecen una gran libertad a la hora de paralelizar diferentes estructuras de datos.

El primer boceto de un módulo para el modelo de islas paralelizaba la población en una RDD, otorgando a cada miembro de la misma un identificador que servía para saber a qué colonia pertenecía cada uno. La idea era agruparlos en base a esa clave y de esa manera gestionar quién se reproducía con quién y cómo se aplicaban las operaciones. Terminamos descartando esta idea ya que, como mencionamos en el apartado 3.4.1, algunos operadores genéticos requieren información sobre la población completa de la colonia. Al estar la población repartida entre diferentes nodos nos hubiéramos visto obligados a agruparla con demasiada frecuencia.

Pasamos pues a una aproximación diferente, que fuerza a la población a quedar agrupada en un mismo nodo: lo que repartimos en el *cluster* es un conjunto de objetos que funcionan como colonias independientes y que pueden gestionar la población a

base de instrucciones del nodo maestro. Su esquema de funcionamiento sería prácticamente idéntico al visto en la figura 5.

Procedemos pues a crear los siguientes módulos:

- **colony.py**: Esta clase recrea el comportamiento de una colonia, desde la inicialización de la población hasta la evaluación y aplicación de los operadores genéticos. Las pruebas a un hilo en local fueron ejecutadas aquí.
- **colony_functions.py**: Esta es una librería que se distribuye entre los nodos del *cluster*. Contiene el código de algunos operadores genéticos ya implementados, con el fin de retirar esa carga del programador si no quiere asumirla y mantener limpio el código anterior.
- **master.py**: Clase que actúa como maestro para el sistema de islas. Gestiona la migración, cache de la RDDs y clusterización en caso de ser necesaria.
- **sparkcolony.py**: Una de las últimas integraciones a este conjunto es esta clase, que recrea el modelo maestro-esclavo visto en el apartado 3.4.1 a partir de una versión de colony.py modificada para distribuir únicamente la evaluación de la colonia. No formaba parte de nuestros planes originalmente, pero nos pareció adecuado comparar también sus resultados.

5.4. Módulos creados

En este apartado explicaremos en mayor detalle los módulos implementados. El único por el que no pasaremos es colony_functions.py, ya que se trata de una librería auxiliar para colony.py y sus funcionalidades están mejor explicadas juntas.

5.4.1. Colony

Diseñamos esta clase con la intención de servir tanto como trabajadores para master.py en el modelo por islas como para ser unidades independientes. Internamente funcionan como lo haría un algoritmo genético convencional como el que vimos en la figura 2, con los operadores descritos en el apartado 3.3.1.

A la hora de inicializarla es necesario proveerla de los siguientes parámetros:

- Función de evaluación, obligatorio.

- Función de generación: el objeto colonia empieza vacío y sus individuos son creados a través de medio cuando el usuario llama a la función *generate*. Este parámetro es obligatorio, aunque alternativamente puede ser reemplazado por un valor entero n, en cuyo caso la colonia generará cadenas de bits de dicha longitud ya sea como lista o como *string*.

- Función de selección: descrita en el apartado 3.3.1, nos permite escoger qué elementos se reproducen entre sí.
  El módulo permite escoger entre selección por ruleta (por defecto), por rango y por torneo de 2; o bien la función que provea el usuario. El elitismo puede ser regulado a través de otro parámetro que veremos más adelante.
  Otra opción disponible es una selección a la que llamamos "pseudo-rango". La idea era producir un efecto similar al rango, pero sin tener que pasar por el proceso de cálculo que implica: sumar todos los valores de la lista, generar un valor al azar dentro de ese intervalo e ir recorriendo la lista sumando evaluaciones hasta superar o igualar esa cantidad.

  $$sum(1..n) = n(n+1)/2 \tag{1}$$

  Este método divide la población en b bloques y asigna a cada b-ésima parte de la colonia una probabilidad de ser escogida en base al sumatorio de sus individuos según la fórmula de la suma acumulada (1) aplicada a las puntuaciones del método de rango. Para escoger un individuo generamos dos valores al aleatorios: uno para saber en cuál de los b bloques hay que buscar, y otro completamente al azar en dicho bloque. Este método es sumamente experimental y, aunque lo mencionamos por haber ofrecido resultados positivos, no podemos considerarlo al mismo nivel que los demás.

- Función de cruce: determina cómo se combinan los individuos para formar nuevos elementos. El usuario puede especificar una función propia o usar cruce en 1 punto (por defecto), en dos puntos o cruce uniforme.

- Función de mutación: esta operación modifica valores al azar de los individuos. Dada su naturaleza es normal que sea el usuario quien la suministre, pero viene una por defecto si estamos empleando cadenas de bits como individuos.

- Función de supervivencia: actúa como la selección, pero sirve para escoger qué elementos de la generación actual pasarán a la siguiente. Puede emplear la definida por el usuario o las mismas descritas en el apartado de selección.

También está la versión simple, que se limita a hacer pasar a los n mejores, pero esta puede resultar excesivamente elitista.

- Ratio de muerte: junto con la función anterior sirve para especificar qué porcentaje de la población es eliminada en cada iteración. Por defecto en 0,85.
- Ratio de mutación: probabilidad individual de cada genoma de mutar. Por defecto en 0,05.

La colonia también puede recibir un objeto de control. Se trata de cualquier tipo de variable que, de ser definida, pasará a ser tratada como otro parámetro de entrada de las funciones de la colonia. El porqué de la existencia de esta particularidad fue una decisión de diseño que describimos con más detalle en el apartado 6.1.3. Dicho brevemente: nos da la posibilidad de guiar los resultados de las diversas operaciones de la colonia con datos adicionales.

Esta clase actúa como un iterable, por lo que podemos acceder a sus elementos igual que haríamos con un vector, aunque no tolera asignación directa.

Las funciones accesibles al usuario son:

- *populate*: inicializa una población de n individuos.
- *get_ best*: devuelve los n mejores individuos (por defecto 1).
- *evolve*: realiza n iteraciones de evolución. Posee dos parámetros opcionales:
  - Objetivo: valor a alcanzar en la evaluación para detenerse. Si no es especificado la colonia evolucionará hasta que termine las n vueltas o se cumpla otra condición de parada.
  - Espera: determina el número de iteraciones que pueden transcurrir sin mejora antes de detener el proceso. Esta opción tampoco está activa por defecto.

5.4.2. Spark Colony

Paraleliza la evaluación de los elementos y la mutación de los nuevos individuos, recreando el modelo maestro-esclavo descrito en el apartado 3.4.1.

Esta clase fue creada modificando el código de la clase Colony, de manera que hiciera uso de RDDs en lugar de listas convencionales, por lo que desde el punto de vista del usuario la única diferencia de uso es que hay que pasarle el Spark Context como parámetro de entrada. El motivo por el que aparece descrita de forma independiente es que, teóricamente, se trata de una clase distinta.

5.4.3. Master

Esta es la clase que recrea el funcionamiento del algoritmo genético en islas descrito en el apartado 3.4.3. Dado que actúa de intermediario entre Spark y el programa principal es menester proporcionarle una serie de parámetros al inicializarlo:

- Spark Context: obligatorio ya que sin él literalmente no tendría forma de interactuar con Spark.
- Método de *clustering*: tal y como describimos en el apartado 3.4.3, las migraciones de la población pueden hacerse o bien intercambiando individuos cada cierto número de iteraciones o reorganizando toda la población con alguna función de clasificación, de manera que cada colonia se centre en un subsector del espacio de soluciones. Por defecto esta clase emplea migración convencional, pero el usuario puede escoger k-vecinos más cercanos [43] o agrupación por modelo de mixturas gaussianas [44].
- Migración: esta variable es empleada para definir cuántos individuos son desplazados por colonia en caso de que ese sea el método activo. Su valor por defecto es 10, pero recomendamos asignarlo en base al problema planteado y la relación entre el número de colonias y el volumen de población total. Si tenemos una población dividida entre muchas colonias pequeñas, el copiar a los mismos individuos tantas veces los sobrerrepresentará y puede derivar en elitismo y convergencia prematura.
- Frecuencia de *clustering*: cada cuántas iteraciones tendrá lugar este proceso. Posee un valor por defecto, que es 90, pero al igual que los casos anteriores recomendamos editar esta variable.
- Frecuencia de *cache*: cuando describimos Apache Spark y las RDDs en el apartado 4.3.1 mencionamos que estas hacían uso de evaluación perezosa, y que por tanto no computaban sus transformaciones hasta que estas eran requeridas por alguna instrucción. Esto quiere decir, que dependiendo del número de pasos

que tenga que hacer cada colonia, es muy factible que al pedir la solución alcanzada nos encontremos con que hemos alcanzado el límite de recursividad del lenguaje antes de terminar. Para evitar esto *cacheamos* la población cada n iteraciones, forzando a Apache Spark a realizar todas las operaciones solicitadas hasta el momento y "desplazar.[el] estado de la RDD. Por defecto su valor es 50, ya que por encima corríamos bastante riesgo de provocar fallos, pero con problemas menores deberíamos poder aumentar este valor. El maestro siempre *cachea* los resultados al terminar una petición de iteraciones.

Funciones de agrupación

Ambas operaciones de *clustering*, k-vecinos y mixturas gaussianas, fueron implementadas haciendo uso de las herramientas de MLlib proporcionadas por Spark, para asegurarnos de que están correctamente adaptadas al entorno distribuido. No entraremos a definir estas funciones en profundidad, ya que supondría desviarse demasiado del objetivo de este trabajo, pero sí que convendría ofrecer una breve explicación de las mismas.

Ambos son algoritmos que buscan separar un grupo de elementos entre subconjuntos de rasgos similares, que reciben el nombre de *clusters*. K-vecinos se llama así porque divide los datos en k grupos, buscando minimizar la distancia entre cada elemento con el centro del suyo, como puede apreciarse en la figura 15.

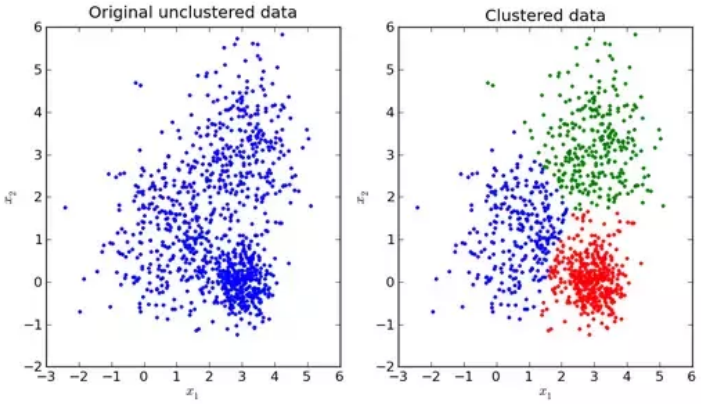

Figura 15: *Clustering* por k-vecinos

Los principales problemas de este sistema son que da lugar a ambigüedades cuando

la forma de los *clusters* no es esférica y que su asignación es rígida, de forma que cada elemento únicamente puede pertenecer a un subconjunto [43].

El modelo de mixturas gaussianas corrige esos inconvenientes gracias a su aproximación probabilística. Se trata de un sistema bastante más complejo, por lo que recomendaríamos encarecidamente al lector acceder a un *paper* más específico si necesita entender los pormenores del mismo [44], que hace uso de cálculo estadístico para determinar el grado de pertenencia de cada elemento a cada clase.

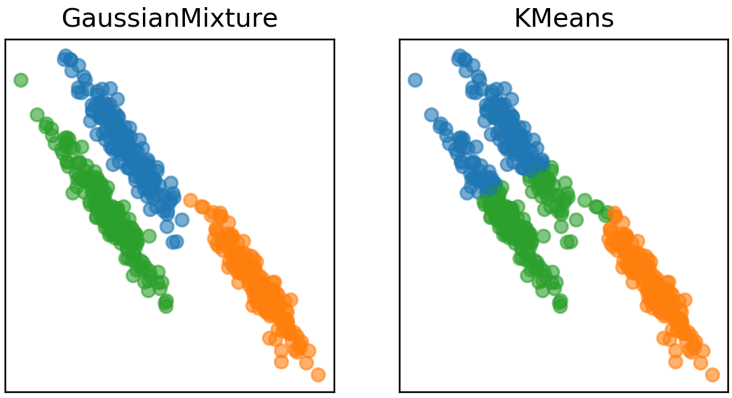

Figura 16: Mixturas gaussianas contra k-vecinos

La figura 16 muestra de forma gráfica el resultado de los modelos descritos frente a la misma distribución, dejando patente la diferencia entre ambos.

Funciones

La clase Master transmite las instrucciones que le sean solicitadas a las colonias que haya distribuido, a la par que se encarga de coordinarlas. Las disponibles al usuario son las siguientes:

- *populate*: Este método genera una serie de colonias con la población y las funciones especificadas por el usuario. Sus parámetros de entrada son los mismos que tendría la inicialización de la clase Colony.
- *get_best*: Devuelve los n mejores elementos generados por el *cluster*, por lo que activarlo supone necesariamente hacer una petición a la RDD.

- *evolve*: Similar al homónimo de la clase Colony, ordena una serie de n iteraciones a las colonias del *cluster*, en las que el usuario puede especificar un objetivo y una espera sin progreso máxima.
  La propia clase se encarga de gestionar la activación de la migración y el *cache* de la población en base a los periodos especificados por el usuario, aunque en caso de necesitar un mayor grado de control también podemos enviar una lista de instrucciones como parámetro de entrada, que serán ejecutadas secuencialmente.

Resumen

En este capítulo hemos explicado que Python, pese a no ser tan rápido como otros lenguajes, ofrece un soporte y comodidad de implementación que lo hacen idóneo para casos en los que necesitamos centrarnos en el propio algoritmo.

A la hora de pedir un trabajo a Spark es necesario crear un entorno virtual en el que encapsular las dependencias del proyecto, que será distribuido entre todos los nodos del *cluster*. Esta forma de proceder ofrece a equipos de varios desarrolladores la posibilidad de coexistir en el mismo entorno sin alterar el espacio de los demás.

Este proyecto ha adaptado el modelo de algoritmo genético paralelo en islas, que son distribuidas como clases de Python gracias a las RDD de Spark. Colony es la clase que actúa como colonia, gestionando las operaciones internas de la misma, y puede actuar de forma independiente o como una isla del modelo distribuido. Master hace la tarea de administrar las colonias gracias a pyspark, y coordina migraciones y procesos de *clustering*.

6 Experimentación

En este capítulo veremos los resultados obtenidos a lo largo del experimento. Comenzamos con un primer apartado que detalla el problema del reparto de paquetes, escogido para poner a prueba el *framework*, y cómo su modelado fue evolucionando hasta dar con una heurística capaz de darle solución.

El segundo apartado describe las métricas empleadas para evaluar el rendimiento de cada modelo, mientras que los resultados se verán relegados en el tercer y último apartado.

6.1. Modelado del problema

Tal y como mencionamos en el apartado 2, el problema empleado para medir los tiempos de cálculo se basa en una aplicación imaginaria para repartir la entrega de una serie de paquetes entre los transeúntes voluntarios de una ciudad. La aplicación deberá tener en cuenta la ruta habitual de cada individuo, el tiempo de llegada a cada de los puntos que transita y el tiempo máximo de espera, respetando siempre dichos elementos.

A fin de trabajar con un escenario realista"hicimos uso de un *script* de Octave capaz de generar un modelo de la malla definida por las conexiones de la red de transporte público de la ciudad de Valencia. Los factores que tiene en cuenta este *script* son la red de metro y el servicio conocido como *Valenbisi*; que consiste en una serie de estaciones en las que los usuarios registrados pueden recoger bicicletas que deberán luego depositar en otra estación similar [45].

Dada la naturaleza de ambos servicios podemos emplear las estaciones a modo de nodos, y los puntos en los que coinciden paradas de metro con puntos de recogida de *Valenbisi* sirven para conectar ambos grafos.

Adaptaremos la idea original de cierta simulación de *Netlogo* [46], un *software* orientado a pruebas con agentes inteligentes, que recrea la ciudad alemana de Salzburgo. Este *paper* sugiere que, a fin de obtener cierto realismo, los desplazamientos de la población pueden ser divididos entre trabajadores, estudiantes y recreativos, quedando fuera de esta clasificación una cantidad reducida de viajeros.

Divide también la ciudad en áreas universitarias, áreas de trabajo y zonas residenciales de forma que los estudiantes se moverán entre zonas residenciales y universitarias, los trabajadores entre residenciales y áreas de trabajo y los recreativos entre zonas residenciales. Cierto porcentaje de viajeros (10 %) no se ceñirá a ningún patrón.

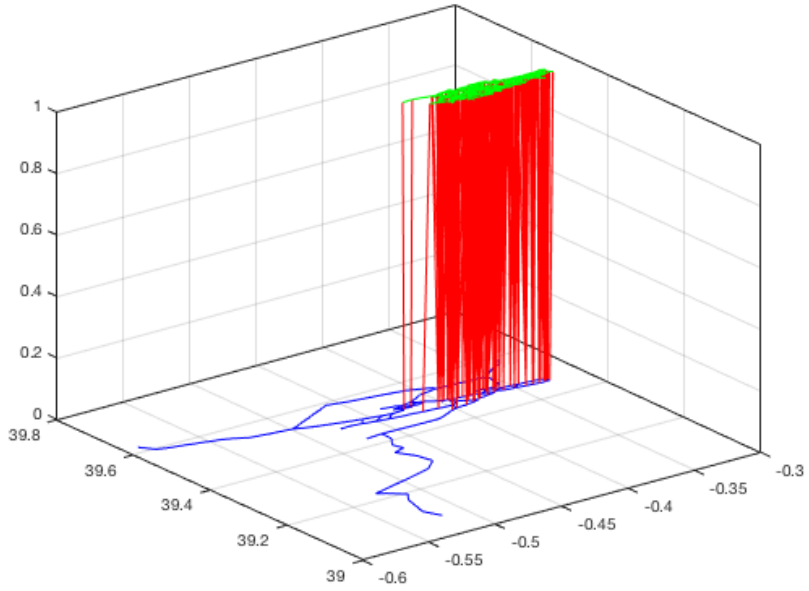

Figura 17: Grafo con la representación de las rutas en metro de Valencia (azul) conectadas con rutas en *valenbisi* (verde)

Tomamos otras dos decisiones de diseño. La primera consiste en limitar el número de paquetes que pueden ser asignados a cada individuo por viaje a 1, esto no solo simplifica considerablemente la resolución del problema sino que además queda coherente con el espíritu de la aplicación, que pretende perturbar el tránsito de los viajeros lo mínimo posible.

La segunda decisión fue tomada poco después a raíz de los resultados más inmediatos, y consistió en restringir las instancias del problema a franjas de unas tres horas. El motivo fue que al permitir la interacción entre usuarios separados por demasiado tiempo estábamos favoreciendo la aparición de múltiples estados imposibles. En la realidad la población también tiende a moverse más a determinadas horas del día (las llamadas "horas punta"), por lo que no creemos que el cambio perjudique especialmente al resultado.

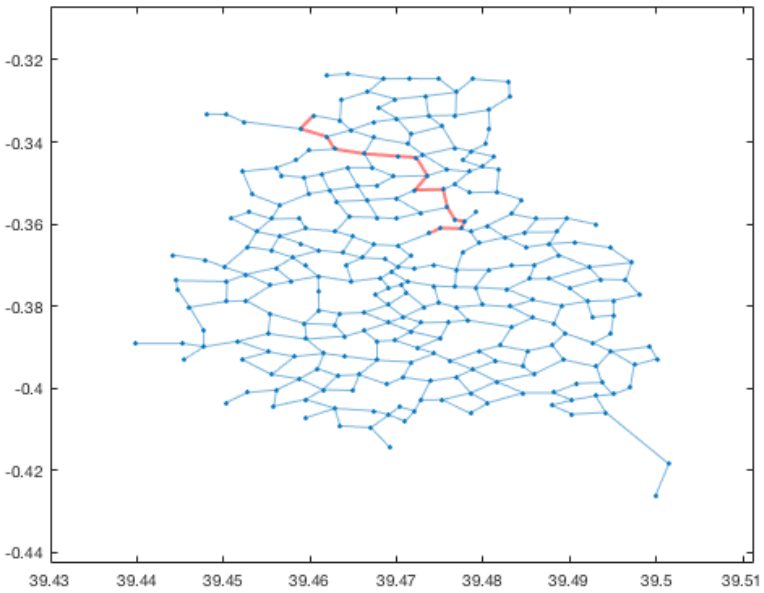

Figura 18: Grafo fusionado de rutas en Valencia. El camino resaltado en rojo muestra en recorrido realizado por un usuario.

6.1.1. Primera versión

Para modelar el problema necesitamos dar con una forma adecuada de representar la información. En una instancia con n viajeros llevando k paquetes tendríamos un vector de longitud n cuyos elementos pueden tomar valores del -1 al $k-1$, de manera que un -1 significa que el n-ésimo viajero no lleva paquetes y cualquier otro de sus posibles valores que dicho viajero lleva el k-ésimo paquete, como refleja el ejemplo de la figura 19.

Escogemos esta representación porque nos permite dividir a los viajeros entre

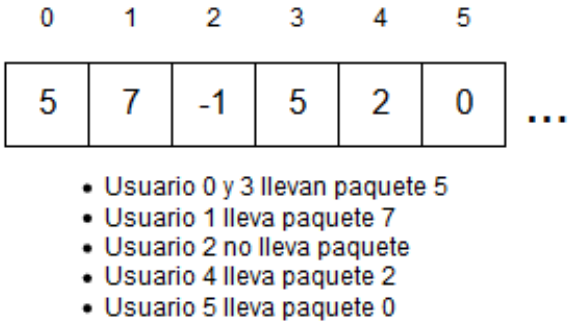

Figura 19: Ejemplo de genoma

potenciales portadores de cada paquete sin forzar un orden concreto en el intercambio del mismo. Como complemento creamos un operador de mutación que, en caso de activarse, tiene un tercio de probabilidades de asignar un -1 al elemento, ya que nos interesa que ese caso no sea infrecuente.

Los viajeros serán representados como un vector de estaciones con tiempos de llegada y salida ordenados en base al recorrido, mientras que los paquetes quedan definidos como un punto de recogida y un punto de entrega.

Camino más corto

Evaluar un cromosoma supone que para cada paquete asignado a algún usuario tendremos que responder a la pregunta de si llega o no y en cuántos saltos lo hace. Si bien un usuario es un vector de estaciones, contestar a estas preguntas implica hacer una búsqueda en el grafo definido por la unión de las rutas de todos los viajeros implicados.

Este grafo variará por cada cromosoma y cada paquete, por lo que solo sabremos que será dirigido con aristas no ponderadas. Desconocemos por tanto la existencia de ciclos y su conectividad, lo cual implica que incluso si los puntos iniciales y finales del recorrido se encuentran en el grafo aún es posible que no exista un camino entre ellos.

Por los motivos citados anteriormente nos es imposible determinar *a priori* si en el recorrido un paso es mejor que otro, lo cual descarta múltiples algoritmos y nos relega a métodos de búsqueda no informados. El método escogido para afrontar este caso es una búsqueda por anchura de profundidad limitada, ya que si se diera el caso de que

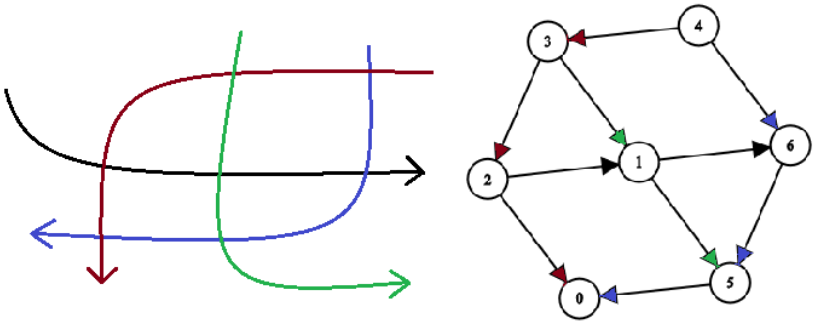

Figura 20: Ejemplo de grafo dirigido definido por 4 viajeros
Esta versión simplificada solo muestra las intersecciones

un paquete tuviera asignada una cantidad demasiado elevada de viajeros queremos evitar que el tiempo de cálculo se vuelva excesivo.

Para evitar tener que hacer dicha operación de búsqueda en todos los casos, antes de empezar nos aseguramos de tener al menos un viajero que pase por el punto inicial y el final, y comprobamos también que no exista ningún viajero que pase por ambos (pudiendo potencialmente llevar el paquete por su cuenta). Un pseudo código que refleja estos pasos puede verse en el algoritmo 1.

Algorithm 1 Camino más corto

Data: V = Conjunto de viajeros
Data: P = Paquete con punto inicial Pi y punto final Pf
begin
  Vi = Subconjunto de V que pasa por Pi
  Vf = Subconjunto de V que pasa por Pf
  if *Vi o Vf son nulos* **then**
    Mejor caso: el paquete no llega
  end
  for *v* **in** *$Vi \cap Vf$* **do**
    if *índice Pi <índice Pf* **then**
      Existe un viajero que puede llevar el paquete solo
    end
  end
  Peor caso: Al menos un usuario recoge y otro entrega el paquete Ejecutamos una búsqueda en anchura
end

Cabe destacar que para este problema el concepto de intersección entre dos caminos no sólo ha de tener en cuenta que coincidan en la misma posición espacial, sino

temporal. La malla se parece más por tanto a un objeto tridimensional (donde en este caso la tercera dimensión es el tiempo) y para comprobar que dos viajeros se cruzan debemos asegurarnos de que la hora de llegada y salida de ambos sea compatible.

Evaluación del cromosoma

El diseño de la función de evaluación supuso otro desafío. Buscamos orientar la evolución de nuestro algoritmo genético de forma que quede repartida la mayor cantidad posible de paquetes usando la menor cantidad posible de viajeros. Tenemos por tanto una variable a maximizar y otra a minimizar, y favorecer una implica necesariamente perjudicar a la otra.

Recordemos que esta es una búsqueda desinformada y no podemos determinar si un camino es mejor que otro. Esto quiere decir que no hay una forma objetiva de saber si un paquete con viajeros asignados que no llega a su destino está en mejor situación que otro de las mismas características, y por tanto no es fácil decidir cómo influye cada uno en la evaluación.

Comenzamos experimentando con variaciones del concepto descrito en el algoritmo 2:

Algorithm 2 Función de evaluación primitiva

```
Data: N = Contribución máxima de un paquete al valor de fitness
Data: P = Conjunto de paquetes
Data: k = Máximo de viajeros esperados por paquete
begin
    fitness = 0
    for p in P do
        v = total viajeros que llevan p
        if p llega then
            fitness += max(N/2, N − N/k ∗ (v − 1))
        else
            fitness += min(N/2, N/k ∗ v)
        end
    end
end
```

La idea tras el pseudo código anterior es la siguiente: buscamos estimular que la

evolución asigne viajeros a cada paquete, para favorecer que eventualmente demos con la combinación adecuada, pero al mismo tiempo queremos favorecer los caminos más cortos, siempre y cuando consigan llevar el paquete a su destino.

Establecemos una aportación individual máxima N y un máximo de viajeros esperado por paquete k; si el paquete no llega aporta un valor al *fitness* que incrementa en función al número de viajeros implicados hasta un límite superior de $N/2$. Cuando el paquete llega aporta un valor a la función de evaluación del N a $N/2$ inversamente proporcional al número de viajeros.

De esta manera nos aseguramos de que un paquete que llega aporte más que uno que no lo hace, pero que el aumento de viajeros repercuta de forma distinta dependiendo de la situación.

Resultado

Las primeras pruebas fueron realizadas localmente con un único hilo, con el objetivo de poner a prueba la heurística empleada. Por ese mismo motivo necesitábamos que el problema fuera resoluble", así que una vez tomados unos viajeros aleatorios los paquetes eran generados de manera que se encontraran en la malla definida por los mismos.

Esta versión inicial devolvió unos resultados poco alentadores: independientemente de los operadores genéticos empleados resultaba demasiado lenta y la convergencia llegaba prematuramente, incluso para tasas de mutación muy elevadas (0.1).

6.1.2. Segunda versión

Después del fracaso de la aproximación original quedó claro que había que resolver los problemas de tiempo de cálculo y convergencia.

Decidimos afrontar primero el problema del tiempo. Uno de los motivos que vuelve tan pesada esta evaluación son las comprobaciones que hay que hacer en cada uno de sus pasos. Sin ir más lejos responder a la pregunta "¿se cruzan estos dos caminos?"supone obtener la unión entre sus dos listas para la cual los tiempos estén

solapados (recordemos que cada uno tiene una holgura) y que además tenga lugar cronológicamente después de la recogida del paquete o el cruce con el transeúnte anterior.

Con el fin de agilizar el cálculo generamos un objeto que, en la inicialización del problema, pre calculará todos los cruces entre todos los viajeros, todos los viajeros que recogen un paquete o pasan por el punto de entrega de alguno y los momentos en que estos sucesos ocurren.

Cualquier información relativa al camino de un viajero que no suponga el cruce con otro punto de interés es prescindible, ya que no nos importa el número de paradas que realiza por el camino, únicamente que se encuentra a una hora en un lugar. La función de evaluación preguntará esos datos directamente y en el caso del sistema distribuido una copia será entregada a cada nodo trabajador, ya que el objeto es inmutable.

Aunque con este cambio no alteramos en absoluto la convergencia, el tiempo de cálculo sí que descendió notablemente, como reflejan los datos de la figura 21

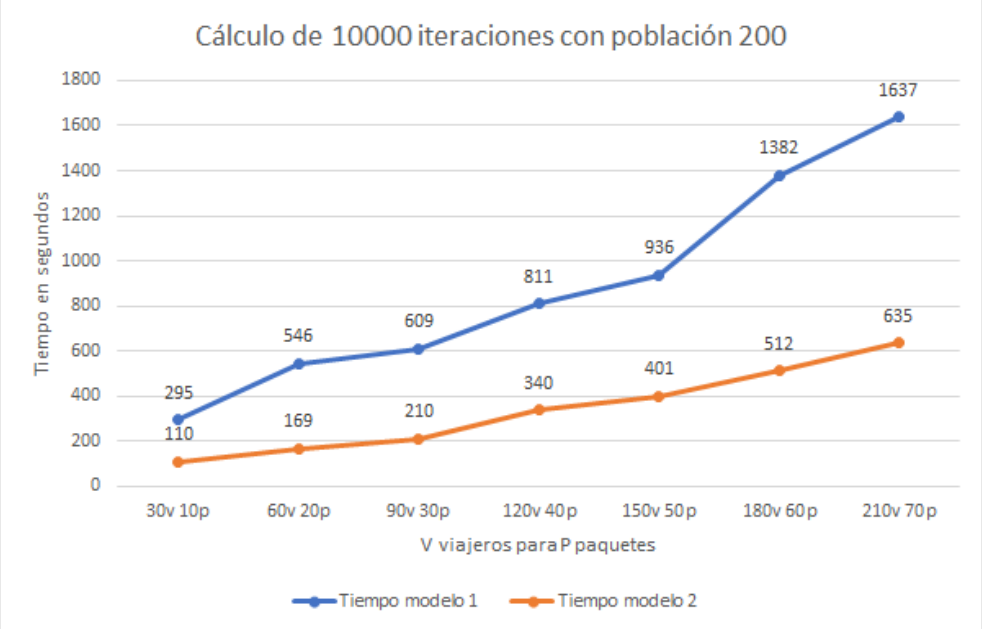

Figura 21: Tiempo promedio para 10 colonias de cada tipo

6.1.3. Tercera versión

Agilizado el cálculo quedaba solucionar la convergencia. La escasa información del problema lo vuelve especialmente enrevesado de guiar, y ello repercute negativamente en el resultado. Diferentes variaciones de la función explicada en el apartado 6.1.1 consiguieron entregar aproximadamente el 50 % de los paquetes con instancias de 100 para 300 viajeros, pero a partir de ese punto el volumen de paquetes descendía drásticamente en una convergencia prematura, hasta menos del 20 %.

Analizando los resultados obtenidos hasta el momento observamos que los mejores elementos de cada población lograban entregar subconjuntos distintos de paquetes. En ocasiones encontramos que dichos subconjuntos no eran incompatibles entre ellos, dándonos a entender que estaban dando con partes distintas de una solución potencialmente parecida. Más tarde, revisando cómo decrecía el tiempo entre iteraciones en base al volumen de datos, decidimos experimentar con una nueva aproximación: en lugar de emplear un algoritmo genético convencional diseñaríamos un modelo que cambia cada x iteraciones.

Dar con un conjunto finito de pasos para solucionar un problema requiere identificar a qué preguntas hay que dar respuesta y en qué orden hay que hacerlo. Esta puede ser una afirmación obvia, pero que es necesario no perder de vista. En nuestro planteamiento original perdimos mucho tiempo tratando de dar con una heurística que resolviera dos preguntas simultáneamente: qué conjunto de viajeros puede llevar cada paquete y cuál es el camino más corto que pueden trazar.

El nuevo modelo busca la respuesta a esas dos preguntas de forma independiente. Tenemos un algoritmo genético que únicamente se encarga de asignar a la mayor cantidad de paquetes posible un subconjunto de viajeros que lo hagan llegar a su destino, sin tener en consideración que el número de viajeros sea mayor del realmente necesario. Cada x iteraciones extraemos el resultado mejor evaluado y lo pasamos por una segunda función que, para cada paquete entregado, busca el camino más corto (en número de intercambios, no distancia recorrida).

A partir de ahí los viajeros prescindibles de cada subconjunto son devueltos a la "pila de cálculo", los datos de los viajeros utilizados son retirados del objeto precalculado descrito en el apartado 6.1.2 y una nueva colonia es generada, reduciendo el espacio de búsqueda. El esquema puede verse descrito como el algoritmo 3. Somos conscientes

de que al hacer esto hemos reducido un problema multi objetivo a uno más simple, pero el objetivo de este trabajo no es probar la efectividad de los modelos distribuidos para este tipo de situación, ese tema ya ha sido abordado en otras investigaciones [21].

Combinamos esta nueva aproximación al problema con una función de evaluación muy similar a la descrita en el algoritmo 2, pero retocada para que penalice los paquetes sin viajeros asignados y premie a los entregados sin bajarles puntuación por número de viajeros, tal y como muestra su pseudo código en el algoritmo 4. Para que sea más agresiva permitimos que esta heurística devuelva valores negativos, y combinamos este cambio con una función de selección por rango, descrita en el apartado 3.3.1.

Algorithm 3 Algoritmo genético segmentado

Data: *obj* = Número de paquetes a entregar
Data: x = Iteraciones de evolución máximas por segmento
Data: w = Iteraciones máximas sin mejora por segmento
Data: C = Objeto de control. Guarda las relaciones entre viajeros y paquetes
begin
  entregados = 0
  while *entregados* <*obj* **do**
    1. Genera una población con los datos de C
    2. Evoluciona colonia con x y w
    3. Actualiza *entregados*, actualiza C
  end
end

Resultado

El nuevo modelo tiene un inconveniente: cada vez que extraemos elementos de la "pila de cálculo.[es]tamos también reiniciando el progreso de la colonia, no obstante descubrimos que la ganancia era mayor que la pérdida. Obtuvimos una evaluación considerablemente mejor que el original evitando los mínimos locales al forzar al algoritmo a progresar. El tiempo de convergencia también es mucho menor, otra consecuencia de decrecer la magnitud del problema con cada iteración.

Algorithm 4 Nueva función de evaluación

```
Data: N = Contribución máxima de un paquete al valor de fitness
Data: P = Conjunto de paquetes
Data: k = Máximo de viajeros esperados por paquete
begin
    fitness = 0
    for p in P do
        v = total viajeros que llevan p
        if v ≠ 0 then
            if p llega then
                fitness += N
            else
                fitness += min(N/2, N/k * v)
            end
        else
            fitness -= N
        end
    end
end
```

El gráfico de la figura 22 muestra los resultados obtenidos en un único hilo comparando este esquema con el modelo secuencial. Este experimento fue realizado comparando la media de 5 colonias de cada tipo, con 500 individuos por colonia, selección por rango y una tasa de extinción del 85 %. El algoritmo segmentado retiraba datos cada 100 iteraciones de la colonia.

En dicha figura podemos apreciar que la versión secuencial, aunque tiene un pico inicial rápido, tiende a estancarse con facilidad. Si la gráfica continuara veríamos cómo va creciendo poco a poco hasta converger en un resultado mucho más favorable (no siempre el 100 % de la entrega), pero no es necesario para ilustrar la abismal diferencia de rendimiento que hay entre los dos modelos.

Hay un detalle que no cambia el resultado global, pero sobre el que queremos llamar la atención: así como en la versión secuencial el tiempo de cálculo permanece, más o menos, estable en todas las iteraciones, en la versión segmentada observamos un descenso paulatino culminado por un aumento considerable cuando tenemos aproximadamente el 85 % de los paquetes entregados. La figura 23 muestra ese contraste.

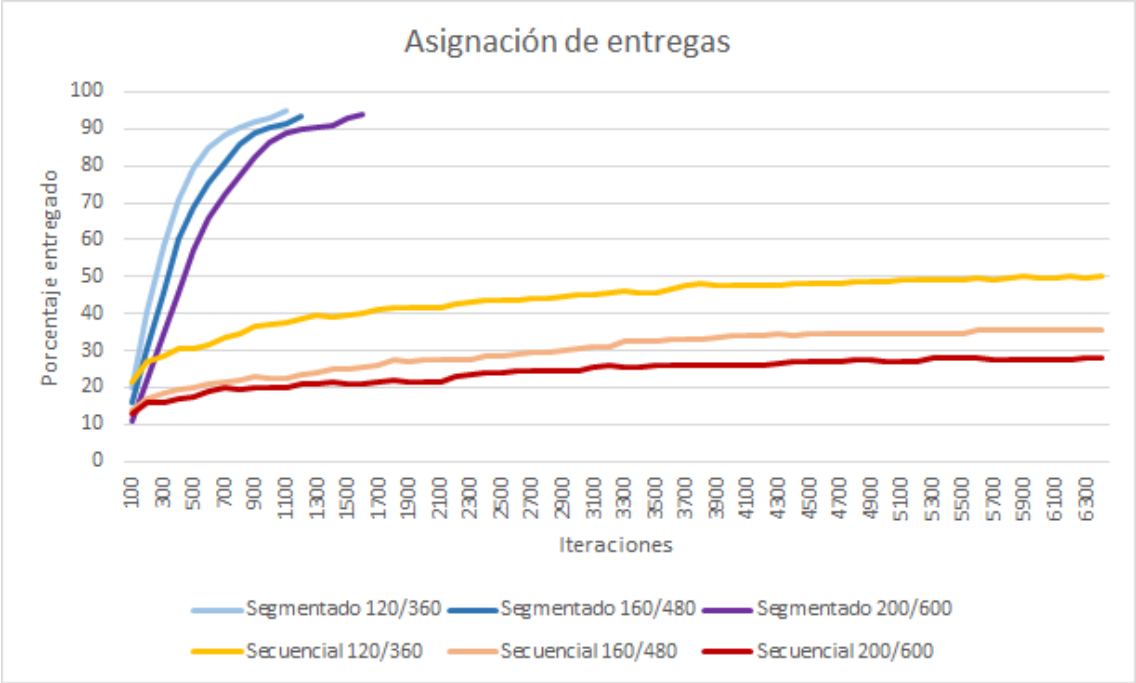

Figura 22: Comparación de convergencia para paquetes/viajeros

Este sistema puede ser fácilmente adaptado a un genético distribuido tanto de modelo maestro esclavo (apartado 3.4.1) como de colonias separadas (apartado 3.4.3). En el primer caso la integración es directa; ya que en el fondo funciona igual que un genético al uso, pero paralelizando el cálculo de algunas operaciones. El segundo caso requeriría que, cada cierto tiempo, el nodo central que gestiona las migraciones actualizara el estado de la evolución y pasara una copia de dicho objeto a todas las colonias.

6.2. Métricas y rendimiento

A la hora de decidir qué métrica emplearíamos para evaluar el resultado de este experimento hemos tenido en consideración dos posibilidades: la primera era emplear el tiempo de ejecución y la segunda el número de iteraciones, ambas hasta alcanzar un valor mínimo en la función de evaluación.

Tomar las iteraciones es más "justo", dado que es independiente de las prestaciones de los equipos usados en las pruebas. Esta métrica ayuda a valorar cómo el tipo

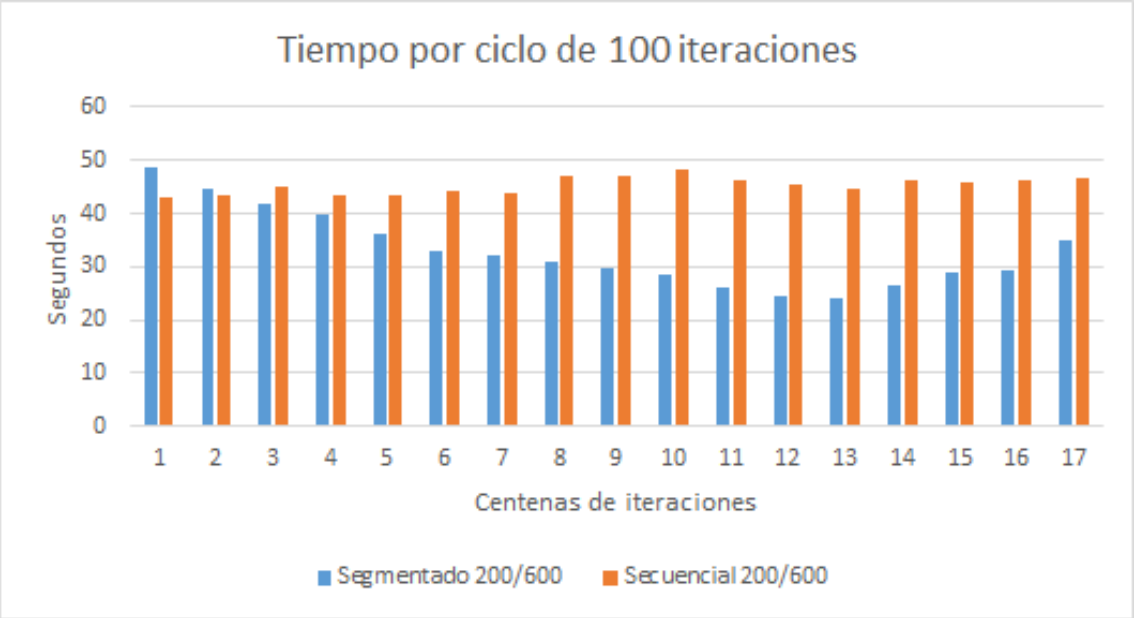

Figura 23: Tiempos por iteración hasta la convergencia del segmentado

de algoritmo influye en la búsqueda y por tanto que se están comportando de la forma deseada. Por otro lado tampoco podemos obviar el tiempo de ejecución, pues se podría argumentar que es preferible una versión que necesite más iteraciones pero que consuma menos tiempo de cálculo del procesador, y no hay que perder de vista que el objetivo último de este trabajo es comprobar si la reducción de tiempo de cálculo supera al sobre coste de la herramienta de *big data* Spark.

Los experimentos consistirán en comparar ejecuciones realizadas con colonias simples a un único hilo, modelo paralelo en islas con migración o *clustering* y paralelo en maestro-esclavo, tomando siempre 5 muestras de cada tipo. En todos los casos hemos obtenido medidas para diferentes volúmenes de población a fin de poder observar el contraste entre velocidad de ejecución y cobertura del espacio de soluciones.

Los operadores genéticos, si no se especifica lo contrario, eran de selección y extinción por rango, cruce en 1 punto, tasa de supervivencia del 15 % y tasa de mutación del 0,8 %. El tipo de *clustering* empleado es mixturas gaussianas, dado que hace una agrupación más lenta pero más precisa que k-vecinos, como explicamos en el apartado 5.4.3. Tanto la migración como la agrupación en estos experimentos tienen lugar cada 100 iteraciones. Por último quisiéramos destacar que los resultados que

veremos a continuación son aquellos que, desde nuestro punto de vista, arrojan algún dato interesante sobre el rendimiento de los modelos.

6.3. Resultados

Las primeras pruebas con el modelo de islas, cuyos resultados aparecen reflejados en la figura 24, mostraron datos prometedores. La ejecución fue realizada para conjuntos de 750 viajeros con 250 paquetes, comparando una población de 200 individuos a un hilo contra varios conjuntos de 3 poblaciones de 200 cada una. Escogemos ese número de poblaciones porque, como mencionamos en el apartado 2, nuestro *cluster* posee un maestro y tres trabajadores.

Los modelos de islas comparados fueron tres: con *clustering*, migración de 10 individuos y una tercera versión sin comunicación entre las colonias, con el objetivo de comprobar que una posible mejora no viniera únicamente dada por una mayor cobertura del espacio de soluciones. Tanto el *clustering* como la migración tenían lugar cada 100 iteraciones.

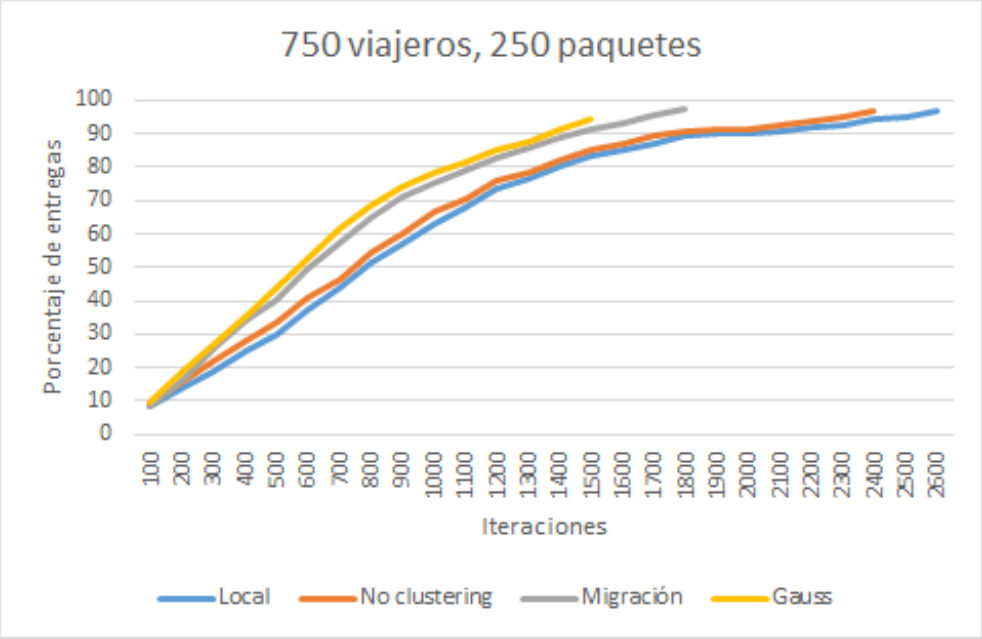

Figura 24: Iteraciones hasta la convergencia

Por los valores obtenidos pudimos comprobar que todas las versiones conseguían

converger en menos iteraciones que el modelo local, como muestra la figura 24. Esto incluye, por muy poco, a la versión sin *clustering* ni migraciones, aunque atribuimos esa ligera diferencia únicamente a una mayor población (si bien es cierto que no está funcionando como una colonia de 600 individuos).

En vista a estos resultados pasamos a experimentar con volúmenes mayores de datos, pero al hacerlo dimos con un problema técnico: al parecer, pese a emplear los métodos nativos de la librería MLlib de Spark, los procesos de agrupación saturaban la memoria de Java (*heap space*) de nuestras máquinas, dificultando el avance del programa y en ocasiones forzándolo a terminar.

Finalmente pudimos experimentar con instancias del problema de 500 paquetes para 1500 viajeros, 1000 paquetes para 3000 viajeros y 2000 paquetes para 6000 viajeros, en todos los casos aumentando el volumen de la población proporcionalmente. La instancia de 2000 paquetes no pudo ser completada con *clustering*, ya que las constantes caídas derivadas del problema de memoria mencionado lo hacían inviable.

La relación entre los resultados ofrecidos se asemejaba a la mostrada en la figura 24, no obstante la versión local con un único hilo siempre daba mejores tiempos que la versión distribuida. Esto se debe a los siguientes motivos:

- A pesar de la paralelización, el tiempo de cálculo por iteración es menor en la versión mono hilo. Esta diferencia no es constante pero debe principalmente a tres factores: el primero es que Apache Spark es el responsable de la distribución de los objetos enviados y no siempre lo hace de forma homogénea. Este modelo de islas puede garantizar que todos los miembros de la misma colonia terminen en la misma máquina, pero no que solo envíen una colonia por máquina.
  El segundo motivo es que hay un sobre coste asociado a las operaciones de envío y recepción de información con el nodo maestro.
  El tercero es que al ser un sistema síncrono el coste temporal de cada conjunto de n iteraciones siempre será el más elevado de entre todas las colonias implicadas.

- La mayoría de los algoritmos de *clustering* pueden dar lugar a grupos de diferente tamaño. Cada uno de estos grupos pasa a ser una colonia independiente que forma parte de un sistema síncrono, de manera que conforme se va descompensando la población cada iteración tarda más.

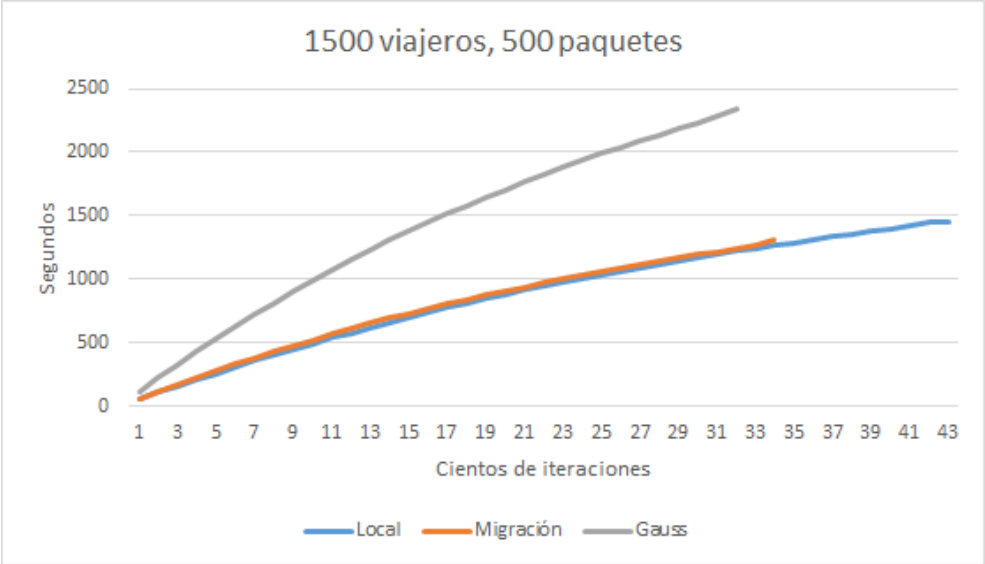

Figura 25: Tiempo en la n-esima iteración

La figura 25 muestra un ejemplo de este fenómeno para instancias de 500 paquetes para 1500 viajeros con población local de 200 frente a modelo en islas de tres colonias de 200 cada una. Esta gráfica no tiene en cuenta el tiempo consumido por los intercambios de individuos ni las operaciones de *clustering*. Vemos que el tiempo de la versión local y las islas con migración evoluciona de forma similar, pero en la *gaussiana* escala de forma distinta: de manera que, aunque necesita menos iteraciones porque funciona mejor a nivel lógico, termina de calcular después de la local.

- Redistribuir elementos es una operación especialmente pesada para Apache Spark, diseñado con la filosofía *write once, read many*. Como consecuencia directa tanto las operaciones de *clustering* como de intercambio de individuos consumen un tiempo en ocasiones mayor que el propio algoritmo genético. La figura 26 muestra las mismas pruebas que la figura 25, pero en esta ocasión añadiendo ese tiempo adicional. Hemos incluido también las medidas para una población de 600 a un único hilo, en las mismas condiciones que los demás grupos.

  La versión en un hilo de 600 no solo necesita menos iteraciones, pues cubre el mismo volumen del espacio de soluciones en una única población conjunta, sino que consume menos tiempo de procesador que las dos versiones en isla para alcanzar la misma calidad de respuesta. En este caso la *clusterización* está

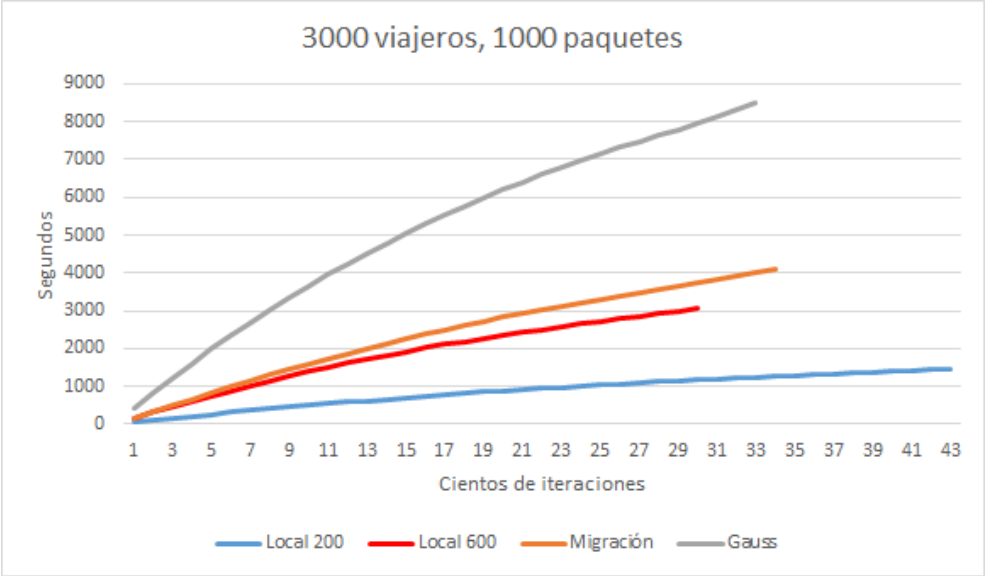

Figura 26: Tiempo con intercambios

consumiendo aproximadamente un 71 % del tiempo de cálculo, y el intercambio de individuos un 65 %, de forma que incluso la versión de 200 individuos gana en tiempo, aunque a nivel lógico consuma más iteraciones.

Llegados a este punto quedaba por ver si existía un volumen de datos para el cual estos modelos de genético distribuido resultaran beneficiosos. El tiempo consumido por la agrupación *gaussiana* parece escalar en relación al volumen y la dimensionalidad de los datos de la población, manteniéndose en torno a la cifra que hemos visto antes.

El tiempo consumido por la migración parece escalar a la par. Experimentando con distintos operadores genéticos llegamos a encontrar configuraciones para las cuales esta proporción era reducida hasta casi el 25 % del total. En este caso específico hicimos uso de selección por ruleta para reproducción y supervivencia, una heurística más costosa que devuelve la gráfica de la figura 27. Pese a esta mejora, una versión local a un único hilo con una población menor devolvió un resultado de igual calidad en menos tiempo.

Las últimas pruebas se centraron en el modelo maestro-esclavo del apartado 3.4.1, por lo que contrastamos el rendimiento del último módulo con el obtenido por colonias del mismo tamaño en un único hilo de procesador. La paralelización actúa enviando

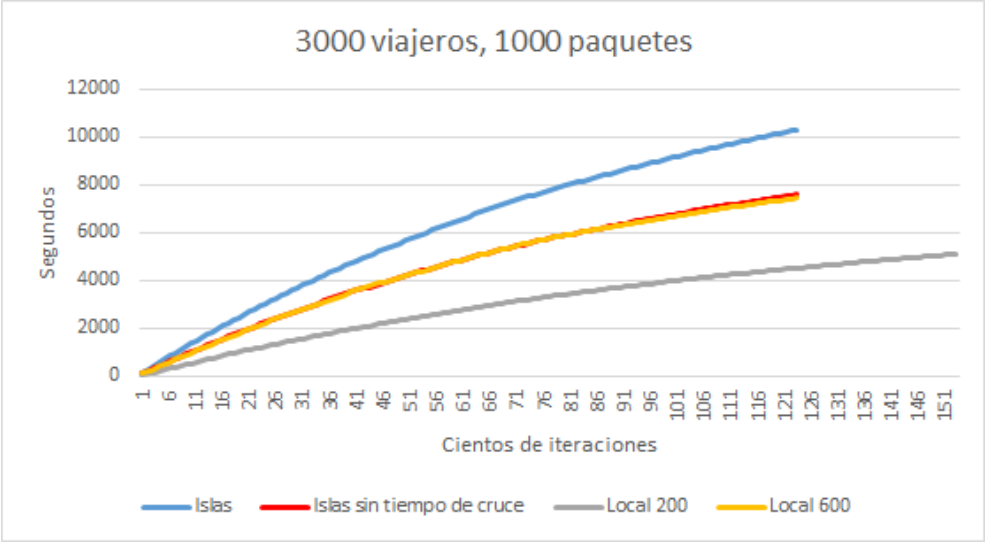

Figura 27: Comparación de Spark con y sin tiempo de cruce

la población al *cluster* para calcular la evaluación y mutación de los individuos, que son las operaciones más costosas que no requieren información de otros miembros de la colonia.

De acuerdo con los resultados el sobre coste de tiempo por operaciones de comunicación se vio disparado considerablemente. La figura 28 muestra un ejemplo de los mismos, comparando el rendimiento en local con 1 hilo de ejecución frente a Spark con los mismos volúmenes de población. En ambos casos la relación tiempo-iteraciones entre la versión con 600 habitantes y con 200 es similar, pero ambas versiones locales son más rápidas que las versiones de Spark, incluso cuando la cobertura del espacio de soluciones es menor.

A partir de este punto experimentamos con magnitudes mayores de datos incrementando la dimensionalidad de los individuos, pero los tiempos de transferencia parecían escalar proporcionalmente a los mismos, de manera que obtenemos un mejor resultado en local, como podemos ver en la figura 29. Respecto al volumen de la población, hasta donde hemos llegado (10000 individuos con 3000 paquetes), obtenemos mejores tiempos en versión local con una población menor, pese a que el número de iteraciones necesarias sea prácticamente el doble. No descartamos que estos últimos resultados sean debidos a la heurística descrita en el apartado 6.1.3.

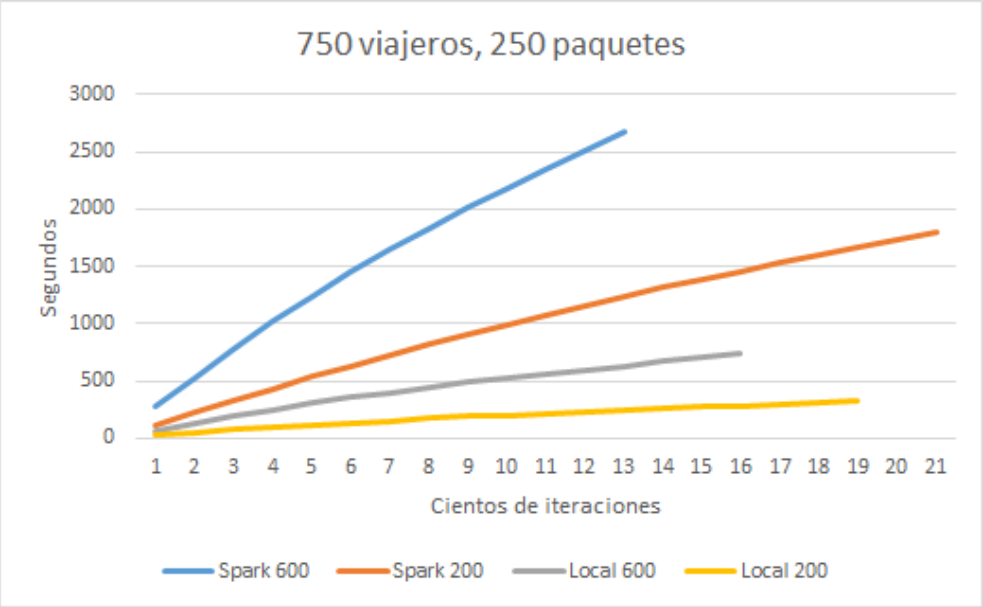

Figura 28: Spark maestro-esclavo contra Local

Resumen

Este capítulo ha servido como síntesis de la experimentación realizada en este proyecto. Comenzamos por hablar del modelado de la entrega de paquetes: un problema de elevada complejidad debido a la forma en que escala el tiempo de ejecución en base al volumen de datos.

Inicialmente tratamos de diseñar una heurística que fuera capaz de asignar portadores a cada paquete a la par que minimizaba el número de intercambios necesarios para llevar a cabo la entrega. Después de varios prototipos infructíferos y tras observar el comportamiento del algoritmo llegamos a un modelo nuevo: cuya estrategia consiste en ir retirando paulatinamente parte de la información para facilitar un cambio en la población. Esta nueva versión superó a las anteriores tanto en número de iteraciones hasta la convergencia como en calidad del resultado obtenido.

Las métricas empleadas fueron las iteraciones hasta la convergencia y el tiempo de ejecución, este último nos interesa a pesar de su ambigüedad porque pretendemos dilucidar si la paralelización con Apache Spark es beneficiosa en las pruebas planteadas.

Los resultados parecen apuntar a que Apache Spark no es la herramienta más ade-

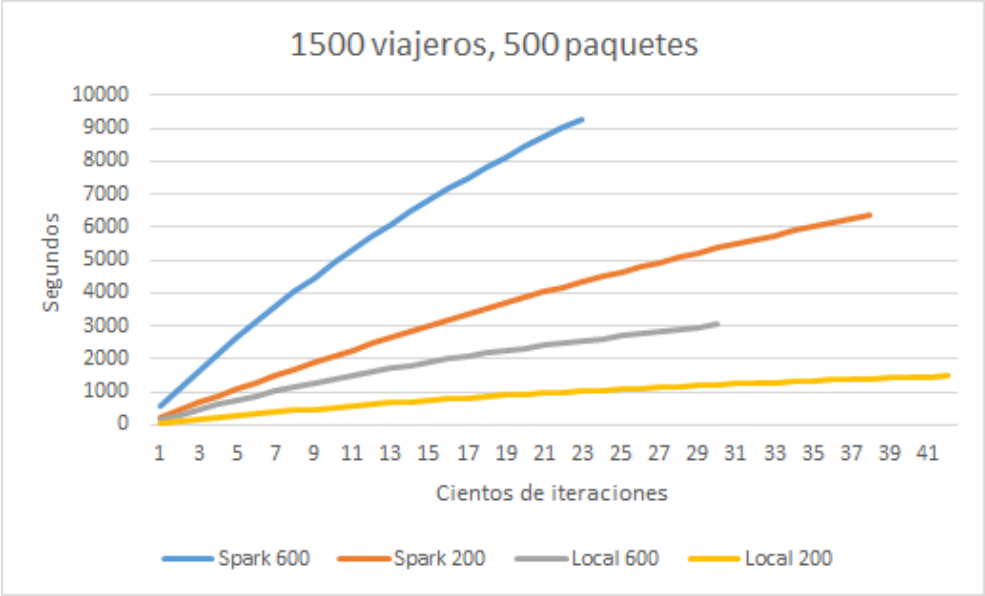

Figura 29: Spark maestro-esclavo contra Local. Nótese la similitud a la figura 28

cuada para este tipo de trabajos, al menos si hablamos estrictamente de rendimiento, pero esta discusión ha sido desarrollada en el capítulo 7.

7 Discusión y conclusiones

El resultado final de este experimento parece haber sido, en base a los tiempos obtenidos, negativo. Pese a todo no hay que descartar que varias de las circunstancias en las que nos encontrábamos pueden habernos llevado a esta conclusión, sin ir más lejos el problema elegido: el reparto de paquetes en transporte público.

Tal y como sugerimos en el apartado 6.1.3, el reparto entre viajeros puede ser considerado un problema multi objetivo con una función a maximizar y otra a minimizar: el total de paquetes entregados y el número de intercambios que necesita cada recorrido. La opción más sencilla hubiera sido crear varias colonias y dotarlas de diferentes funciones de evaluación, pero hacerlo implicaba usar también una función distinta para el algoritmo genético mono hilo, dificultando la comparación del rendimiento, que es el objetivo final de este trabajo.

La heurística planteada es capaz de devolver una respuesta satisfactoria con una función de evaluación simple, pero su forma de proceder beneficia considerablemente a una colonia ligera capaz de hacer más iteraciones rápidas, eliminando viajeros y paquetes con mayor frecuencia. Esto quedó demostrado en el apartado 6.3, donde la colonia mono hilo con menor población siempre obtenía mejor tiempo. No consideramos este punto como algo especialmente negativo, pues refleja que el diseño de la estrategia del algoritmo puede ser tan relevante como la propia potencia de cálculo.

El segundo factor decisivo es que el *cluster* al que tuvimos acceso para la realización de estas pruebas es una versión virtualizada de un sistema distribuido real. Esto acarrea una pérdida de rendimiento que puede haber derivado en los elevados tiempos de intercambio de información, y que lamentablemente no podemos medir por no tener acceso a un equipo más adecuado.

Apache Spark puede ser configurado para recrear varias aproximaciones al concepto de algoritmo genético distribuido, como lo son el modelo maestro-esclavo o el de

islas, pero no está optimizado para este tipo de operaciones que requieren escrituras muy frecuentes. Este problema es especialmente acuciante en el caso del modelo maestro-esclavo, donde el sobre coste de tiempo escalaba siempre a la par que lo hacía el volumen de datos. También hemos observado que la capa de abstracción que ofrece Spark puede actuar en detrimento del algoritmo, ya que necesitaríamos cierto grado de control sobre la carga de trabajo de los nodos, especialmente cuando el sistema es síncrono y las colonias cambian de tamaño.

Los *frameworks* para algoritmos genéticos distribuidos han dado resultados positivos en investigaciones previas, especialmente en el modelado de funciones multi objetivo [21], pero cuando son combinados con herramientas de *big data* es preferible que además manejemos un volumen de datos elevado e irreductible, de manera que una sola máquina sea incapaz de mantener dicha información en memoria. Si estas circunstancias no se cumplen, en cuestión de rendimiento, quedarán por detrás de la misma instancia del problema alojada en una única máquina.

8 Trabajo futuro

Es natural que una investigación se ramifique durante su desarrollo y esta no es una excepción. Conforme trabajamos en este proyecto nos surgieron dudas e inquietudes que o bien estaban fuera de nuestro objetivo fijado o bien tuvieron que quedarse en el tintero por falta de tiempo.

En este capítulo explicamos algunas de esas ideas y los motivos por los que las consideramos interesantes cara a una futura investigación.

Grafos ponderados para el problema del reparto

El procedimiento diseñado para el problema del reparto de paquetes probó ser capaz de encontrar soluciones en un tiempo aceptable, casi podría decirse que mejor que la cobertura ofrecida por Spark, pero fue adaptado porque teníamos muy poca información a la hora de resolver el problema.

Nuestra propuesta sería buscar una heurística que ponderara los caminos recorridos por los usuarios. Esto serviría como referencia a la evaluación a la hora de juzgar en qué situación está cada paquete, facilitando el uso de una función multi objetivo en condiciones. Lamentablemente no podemos orientar más al lector en este aspecto, solo indicar que en su momento el tiempo y la distancia recorrida no nos parecieron medidas adecuadas, debido a la naturaleza variante del grafo que describen los viajeros asignados.

Modelo de islas en Docker

Docker es una tecnología de *clustering* de código abierto diseñada para automatizar el despliegue de aplicaciones. En la industria es empleada como medio para la integración continua de proyectos por su ligereza, facilidad de uso y escalabilidad. Docker funciona desplegando contenedores (*dockers* en inglés), que son unidades virtuales de sistemas operativos (generalmente Ubuntu y derivados) capaces de llevar a cabo

diferentes tareas sin el lastre de dedicar tiempo a elementos prescindibles, como lo sería una interfaz gráfica.

En lugar de intentar adaptar una herramienta como Apache Spark, el modelo de islas seguramente funcionaría mejor en un entorno con un comportamiento similar: la idea es diseñar una imagen de máquina virtual con los requisitos necesarios para el algoritmo, que luego sería desplegada por Docker en modo de enjambre entre múltiples máquinas.

Cada contenedor funcionaría como una colonia independiente, que realizaría iteraciones de evolución puntuadas por envíos y recepción de individuos con un nodo maestro.

Colonias asíncronas

Si bien mencionamos en el apartado 3.3 que los algoritmos genéticos no pretenden ser una representación fidedigna de la evolución, también es cierto que algunos estudios han experimentado con otros elementos implicados en procesos reproductivos como pueda ser la edad del individuo, obteniendo resultados prometedores [12].

En este caso experimentaríamos con la migración asíncrona entre colonias distintas, destinadas a optimizar diferentes partes de un problema multi objetivo. El estudio podría comparar los resultados de varias topologías, añadiendo migraciones dirigidas entre colonias con funciones de evaluación progresivamente más complejas.

Una opción distinta para añadir realismo a la migración sería introducir el concepto de "preferencia", es decir, que en un grafo completo los individuos viajaran a una colonia u otra en base a alguna heurística predefinida, como la distancia al centro del grupo.

Las colonias evolucionarían por su cuenta y cada cierto tiempo enviarían miembros de su población a las máquinas a las que estuvieran dirigidas o a un nodo maestro, mientras que la llegada de individuos podría gestionarse mediante un hilo dedicado a un *buffer* de entrada. Naturalmente Apache Spark es menos que idóneo para este trabajo, pero tal vez un enjambre de máquinas virtuales como las del mencionado Docker podrían reproducirlo eficazmente.

Bibliografía

[1] L. J. Eshelman, T. Bäck, D. Fogel, and T. Michalewicz, "Genetic algorithms," *Evolutionary Computation*, vol. 1, pp. 64–80, 2000.

[2] K.-S. Tang, K.-F. Man, S. Kwong, and Q. He, "Genetic algorithms and their applications," *IEEE signal processing magazine*, vol. 13, no. 6, pp. 22–37, 1996.

[3] R. Paz Hernandez, *Optimización y experimentación del diseño con algoritmos genéticos y metamodelos en la minimización del peso en piezas obtenidas mediante fabricación aditiva*. PhD thesis, 2014.

[4] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms.," in *IJCAI*, vol. 89, pp. 762–767, 1989.

[5] R. Tanese, "Distributed genetic algorithms for function optimization," 1989.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *Usenix Login*, vol. 37, no. 4, pp. 45–51, 2012.

[7] W. Dean, "Computational complexity theory," 2015.

[8] S. B. Cooper, *Computability theory*. Chapman and Hall/CRC, 2017.

[9] I. H. Osman and J. P. Kelly, "Meta-heuristics theory and applications," *Journal of the Operational Research Society*, vol. 48, no. 6, pp. 657–657, 1997.

[10] K. M. Weiss, "Evolution," *The International Encyclopedia of Biological Anthropology*, pp. 1–16, 1998.

[11] L. Davis, "Handbook of genetic algorithms," 1991.

[12] G. S. Hornby, "Alps: the age-layered population structure for reducing the problem of premature convergence," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 815–822, ACM, 2006.

[13] L.-C. Hsu, "Forecasting the output of integrated circuit industry using genetic algorithm based multivariable grey optimization models," *Expert systems with applications*, vol. 36, no. 4, pp. 7898–7903, 2009.

[14] K. C. Chan, P. C. Hui, K. Yeung, and F. S. Ng, "Handling the assembly line balancing problem in the clothing industry using a genetic algorithm," *International Journal of Clothing Science and Technology*, vol. 10, no. 1, pp. 21–37, 1998.

[15] S. Varun Kumar and R. Panneerselvam, "A study of crossover operators for genetic algorithms to solve vrp and its variants and new sinusoidal motion crossover operator," *Int. J. Comput. Intell. Res*, vol. 13, no. 7, pp. 1717–1733, 2017.

[16] M. Srinivas and L. M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[17] Z. Michalewicz and C. Z. Janikow, "Handling constraints in genetic algorithms.," in *ICGA*, pp. 151–157, 1991.

[18] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Evolutionary computation 1: Basic algorithms and operators*. CRC press, 2018.

[19] T. C. Fogarty and R. Huang, "Implementing the genetic algorithm on transputer based parallel processing systems," in *International Conference on Parallel Problem Solving from Nature*, pp. 145–149, Springer, 1990.

[20] R. Hauser and R. Männer, "Implementation of standard genetic algorithm on mimd machines," in *International Conference on Parallel Problem Solving from Nature*, pp. 503–513, Springer, 1994.

[21] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "A study of master-slave approaches to parallelize nsga-ii," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[22] U. Kohlmorgen, H. Schmeck, and K. Haase, "Experiences with fine-grainedparallel genetic algorithms," *Annals of Operations Research*, vol. 90, pp. 203–219, 1999.

[23] S. Baluja, "Structure and performance of fine-grain parallelism in genetic search.," in *ICGA*, pp. 155–162, 1993.

[24] J. Sarma and K. De Jong, "An analysis of the effects of neighborhood size and shape on local selection algorithms," in *International Conference on Parallel Problem Solving From Nature*, pp. 236–244, Springer, 1996.

[25] D. Whitley, S. Rana, and R. B. Heckendorn, "Island model genetic algorithms and linearly separable problems," in *AISB International Workshop on Evolutionary Computing*, pp. 109–125, Springer, 1997.

[26] S. J. Gould and N. Eldredge, "Punctuated equilibria: the tempo and mode of evolution reconsidered," *Paleobiology*, vol. 3, no. 2, pp. 115–151, 1977.

[27] E. Alba and J. M. Troya, "Analyzing synchronous and asynchronous parallel distributed genetic algorithms," *Future Generation Computer Systems*, vol. 17, no. 4, pp. 451–465, 2001.

[28] R. Tanese, "Parallel genetic algorithm for a hypercube," in *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*, Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987.

[29] E. Cantú-Paz, "Topologies, migration rates, and multi-population parallel genetic algorithms," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pp. 91–98, Morgan Kaufmann Publishers Inc., 1999.

[30] T. C. Belding, "The distributed genetic algorithm revisited," *arXiv preprint adap-org/9504007*, 1995.

[31] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[32] "The web is much bigger (and smaller) than you think." `https://www.forbes.com/sites/ciocentral/2012/04/24/the-web-is-much-bigger-and-smaller-than-you-think/#218755a97619`. Accessed: 2019-05-10.

[33] "How much data do we create every day?." `https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#2605893260ba`. Accessed: 2019-05-08.

[34] A. S. Foundation, "Apache Hadoop documentation," 2006.

[35] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[36] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using mapreduce," in *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pp. 13–18, IEEE, 2009.

[37] C. Jin, C. Vecchiola, and R. Buyya, "Mrpga: an extension of mapreduce for parallelizing genetic algorithms," in *2008 IEEE Fourth International Conference on eScience*, pp. 214–221, IEEE, 2008.

[38] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 785–793, IEEE, 2012.

[39] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. .ºReilly Media, Inc.", 2015.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[41] "Developer survey results." `https://insights.stackoverflow.com/survey/2019#technology`. Accessed: 2019-06-20.

[42] "2018 developer skills report." `https://research.hackerrank.com/developer-skills/2018/`. Accessed: 2019-06-21.

[43] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[44] G. J. McLachlan and K. E. Basford, *Mixture models: Inference and applications to clustering*, vol. 84. M. Dekker New York, 1988.

[45] "Valenbisi." `http://www.valenbisi.es/`. Accessed: 2019-06-15.

[46] G. Wallentin and M. Loidl, "Agent-based bicycle traffic model for salzburg city," *GI_ Forum J. Geogr. Inf. Sci*, vol. 2015, pp. 558–566, 2015.