Grado en Ingeniería Informática

Composición de servicios con restricciones usando utilidad

Utility-based service composition with constraints

Realizado por
Katia Moreno Berrocal

Tutorizado por
Francisco Javier Durán Muñoz

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio de 2022

Resumen

La composición de servicios puede ser entendida como la correspondencia servicio-proveedor dentro de una arquitectura basada en servicios. No solamente bastará con realizar una mera asignación, sino que dicha selección deberá ser la óptima atendiendo a la arquitectura y a las limitaciones impuestas por el usuario. Este problema comenzará a ser intratable a medida que el número de proveedores y servicios aumenta, creciendo su complejidad a un ritmo exponencial.

En este trabajo se han planteado el uso de dos procedimientos atendiendo al problema a resolver, dando como resultado una composición más que aceptable en un tiempo muy razonable. El primero de ellos será aplicado cuando se busca la composición óptima sin tener en cuenta ninguna clase de restricción y la segunda, y más general, es el procedimiento en el que dichas restricciones sí se tienen en cuenta. Otra novedad que se presenta es que se ha podido resolver la composición con casuísticas que hasta el momento eran intratables mediante el procedimiento en el que está basado este trabajo.

Palabras claves: Composición de Servicios, Algoritmos Genéticos, Heurísticos, Restricciones

Abstract

Service composition can be understood as the service-provider correspondence in a service-based architecture. Not only a mere mapping will be enough, but the selection must be optimal in terms of the architecture and the constraints imposed by the user. This problem will become intractable as the number of providers and services increases, growing in complexity at an exponential growth rate.

In this work we have proposed the use of two procedures according to the problem to be solved, resulting in a more than acceptable composition in a very reasonable time. The first of them will be applied when the optimal composition is searched for without taking into account constraints, and the second, and more general, is the procedure in which these constraints are taken into account. Another novelty that is presented is that it has been possible to solve the composition under circumstances that until now were intractable by using the procedure on which this work is based.

Keywords: Service Composition, Genetic Algorithms, Heuristics, Constraints

Índice general

1. Introducción **7**

1.1. Motivación 9

1.2. Objetivos 9

1.3. Organización 10

2. Composición de servicios **13**

2.1. ¿Qué es la composición de servicios? 13

2.2. Modelo arquitectónico 14

2.3. Atributos de calidad de servicio 17

2.4. Estado del arte 20

3. Conocimientos previos **27**

3.1. Introducción al método de utilidad 27

3.2. Adaptaciones necesarias 29

3.3. Algoritmos genéticos 32

4. Métodos basados en la utilidad **37**

4.1. El método de utilidad 37

4.2. El método de utilidad sin restricciones 38

4.3. El método basado en la utilidad con restricciones 40

4.3.1. Primera fase: pre-cómputo 41

4.3.2. Segunda fase: Aplicación del algoritmo genético 43

4.3.3. Tercera fase: Conversión cromosoma - composición 45

4.4. Grados de utilidad óptimos 46

5. Resumen de la implementación **49**

6. Experimentos **53**

6.1. Experimento 1 54

6.2. Experimento 2 57

6.3. Discusión de los experimentos 60

7. Conclusiones y líneas futuras **63**

CAPÍTULO 1

Introducción

La composición de servicios es un tema que ha sido ampliamente debatido por la comunidad científica y que, además, ganó mucha popularidad en el ámbito de los desarrolladores, técnicos y empresas debido a su gran versatilidad.

La unidad o entidad más básica de la composición de servicios es el *servicio*. Un servicio es una entidad que se encarga de realizar cierta tarea. Entiéndase como tarea, por ejemplo, el sistema para efectuar una consulta del pronóstico del tiempo. A medida que el número de las aplicaciones basadas en microservicios crecen, sumado a que estamos ante un inminente auge de la tecnología IoT, y la creciente complejidad de las aplicaciones, entre otras cosas, el número de servicios web ha aumentado dadas las facilidades que estos proporcionan en dichos campos.

El orden de ejecución de los servicios, atendiendo a sus dependencias y recursos, creará diferentes soluciones. La descripción de este orden de ejecución se puede entender como una arquitectura. Véase la figura 1.1. En ella podemos ver la especificación visual de una arquitectura en la cual los círculos son los servicios a ser ejecutados, las flechas son los flujos del orden de ejecución y los rombos las estructuras de control. Estos conceptos serán explicados en mayor grado de detalle en los siguientes capítulos.

El servicio deberá ser hospedado en una máquina en concreto. Esta es denotada como *proveedor*. La oferta de proveedores no es única y, cada uno de ellos ofrecerá

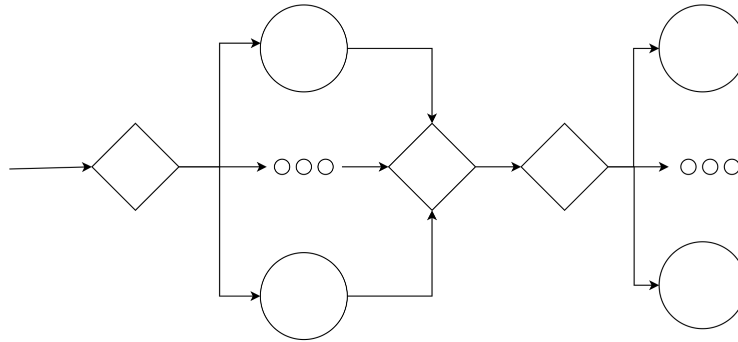


Figura 1.1: Representación gráfica de una arquitectura basada en servicios.

una serie de atributos que hará que el usuario se decante más por una opción antes que por otra. Estos atributos son los llamados *atributos de calidad de servicio*. La elección de un proveedor para cierto servicio es la llamada composición. Atendiendo a los proveedores seleccionados, el valor de los atributos de calidad de servicio cuando se agregan para obtener el valor de la aplicación completa, pueden cambiar radicalmente.

La composición puede adaptarse a las necesidades del usuario, pudiendo dar más preferencia a unos atributos que otros, esta importancia vendrá denotadas por los *pesos asociados a los atributos*. También será posible poner valores máximos o mínimos a los que deben adaptarse los valores de los atributos de la composición dada. Estas condiciones son las llamadas *restricciones*.

En este trabajo daremos solución al problema de la composición de servicios mediante dos enfoques.

- Uno con el uso de un heurístico, que junto a un algoritmo genético, darán una solución más que suficiente en un tiempo muy razonable cuando se tratan restricciones. Este método también podrá dar una solución pseudo-óptima cuando no se tengan restricciones. En este caso, buscará la mejor composición posible a nivel global.

- El segundo método no tiene en cuenta dichas restricciones, por lo que la selección se realizará a nivel local.

1.1. Motivación

Como se verá en secciones posteriores, el tiempo de seleccionar la mejor composición crece exponencialmente a medida que aumentan el número servicios y, más drásticamente, si aumenta la oferta de proveedores. Existen diversos enfoques que serán explicados posteriormente en la sección 2.4, (*Estado del Arte*). Cada uno de ellos tiene diferentes maneras de tratar el problema.

El procedimiento en el que está basado este TFG es el expuesto por Farhad et al. en [5]. En dicho trabajo, presentan un enfoque que proporciona muy buenos resultados en términos de calidad de solución y tiempo de ejecución.

La principal motivación para realizar este TFG, es el de implementar dicho método y el de solventar el principal problema que se presenta en el mismo: la imposibilidad de dar una composición con atributos dependientes del canal.

Asi mismo, este TFG busca poder dar una composición en un tiempo razonable, con una solución aceptable, en cualquier tipo de arquitectura balanceada y atributo de calidad de servicio tenido en cuenta.

1.2. Objetivos

El objetivo de este TFG es el de aplicar una técnica que sea capaz de aproximar la mejor composición en un tiempo aceptable y que la calidad de la solución sea lo suficientemente buena para ser aceptada. Así mismo, será necesario que dicha técnica pueda aceptar atributos dependientes del canal tales como, latencia o ancho de banda. Para cumplir con estos objetivos, se ha realizado al siguiente división:

1. Creación de un algoritmo que implemente la composición de servicios pseudo-óptima con y sin restricciones.

2. Implementación de un procedimiento que pueda tratar atributos de calidad de servicio hasta ahora no tratados por el método basado en utilidad debido a

sus limitaciones.

3. Creación de mecanismos que valoren la calidad de la solución.

4. Realización de pruebas con exhaustivas métricas para valorar que los resultados son satisfactorios.

5. Comparación de la técnica aplicada con otros dos métodos ampliamente aceptados por la comunidad científica.

1.3. Organización

Para la realización de este trabajo se ha seguido un proceso iterativo incremental. El principal motivo, es que se trataba de un trabajo de investigación en el que se han utilizado ciertos heurísticos nunca antes aplicados y, por tanto, ha sido necesario volver a las primeras fases de análisis cuando no se obtenían resultados satisfactorios.

A grandes rasgos, los bloques en los que se ha organizado este trabajo han sido los siguientes:

1. Búsqueda de información y documentación ($\backsim$ 30h): fase en la que se buscó todo el trabajo relacionado acerca de esta materia. En esta fase fue necesario elegir el método en el que se basó principalmente este trabajo.

2. Análisis del problema ($\backsim$ 15h): se estudió cómo mejorar el problema encontrado en los atributos que no podían ser tratados por las limitaciones.

3. Desarrollo software ($\backsim$ 160h): etapa en la que se programaba la solución a medida que se iban obteniendo los resultados.

4. Pruebas ($\backsim$ 25h): proceso en el que se comprobó el correcto funcionamiento de la aplicación desarrollada.

5. Corrección de errores y casuísticas ($\backsim$ 35h): se ajustaron los hiperparámetros utilizados del los algoritmos utilizados.

6. Realización de los experimentos (∽ 10h): despliegue la aplicación y recolección de datos para diferentes casos.
7. Análisis de los resultados (∽ 10h): síntesis de los resultados de la fase anterior.
8. Documentación de los resultados (∽ 10h): ralización las conclusiones de los datos obtenidos.
9. Revisión de la documentación y correcciones menores (∽ 1h).

La elaboración de este TFG no se realizó de manera lineal. Es decir, no se pasaron por todas las fases y agotaron todas las horas. Entre las etapas 1 y 7 se pudieron volver a pasos anteriores o posteriores debido a la naturaleza de este trabajo.

CAPÍTULO 2

Composición de servicios

En este capítulo se abordará la definición del problema al que estamos dando solución. Para ello, presentaremos una serie de conocimientos de carácter general para entender las decisiones y procedimientos aplicados en el mismo.

2.1. ¿Qué es la composición de servicios?

La composición de servicios puede ser resumida en una correspondencia servicio - proveedor. Véase la figura 2.1. Cada servicio tiene disponible una lista de proveedores potencialmente distinta a la del resto, ya que no necesariamente todos los proveedores pueden ser compatible con el tipo de cómputo que debe realizar el servicio. La composición deseada será la que tenga el menor valor agregado de atributos que tengan que ser minimizados y viceversa. La composición, por ejemplo, de coste, latencia y tiempo de respuesta mínima y de disponibilidad y fiabilidad máxima será la deseada como solución.

Servicio	Proveedor
w_0	p_i
w_1	p_j
...	...
w_n	p_z

Figura 2.1: Representación de una composición

La selección de un proveedor antes que otro es una mera cuestión de preferencias del usuario. Cada proveedor proporciona una serie de atributos de calidad de servicio. Lo que hará que una composición sea mejor que otra es que el valor de cierta función objetivo para una composición concreta sea mayor que la de otra. En las secciones siguientes se verá esta cuestión con mayor detalle. La función objetivo es un mecanismo que nos permite obtener la calidad de la solución proporcionada. Se trata de un valor discreto situado entre 0 y 1, siendo 1 la de mayor calidad posible.

2.2. Modelo arquitectónico

En nuestro trabajo, consideramos como *arquitectura* la descripción de las dependencias y el orden de ejecución de los servicios de la aplicación dada. Tal como se introdujo en la sección 1.1, atendiendo a la manera en la que compongamos los servicios, la aplicación seguirá un comportamiento u otro.

El mecanismo que disponemos para poder añadir este comportamiento es el de la incorporación de los llamados *patrones*. Un patrón se define como el comportamiento que deben seguir todos los servicios e incluso composiciones de patrones dentro del mismo. Para poder delimitar qué servicios están sujetos a este comportamiento, se hace uso de las *estructuras de control*. Véase la figura 1.1 de la sección 1.

Otro concepto importante es el de *componente*. Consideramos que un componente es la encapsulación de todos los servicios dentro de un patrón. A su vez, un componente dentro de un patrón compuesto de más servicios, e incluso componentes, es también considerado componente. La sucesiva aplicación de esta definición finalizará con un único componente que englobará a la aplicación por completo. Este concepto es importante y en la siguiente sección se verá en mayor grado de detalle el motivo por el que se aplica dicha definición. A partir de ahora, y por simplicidad, llamaremos como *nodos* tanto a componentes como a servicios cuando la definición tenga en cuenta estos dos entes a la vez.

En este trabajo se utilizan los principales patrones de diseño definidas por BPMN. Se trata de un estándar para la definición de procesos de negocio, en la que es posible representar de manera gráfica los flujos de trabajo o ejecución. Los patrones utilizados en este trabajo son: *secuencial, paralelo, condicional e iterativo.*

- **Secuencial**. Es el patrón que dado dos nodos a y b, siendo a el nodo anterior y b el posterior, b no puede empezar su ejecución hasta que a no haya acabado la suya. Véase la figura 2.2.

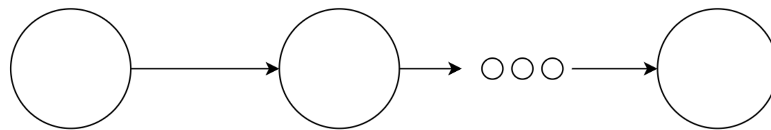


Figura 2.2: Patrón arquitectónico secuencial.

- **Paralelo**. Se dice que un componente se trata de un paralelo, si todos los nodos que los compone se deben ejecutar y no es hasta que todos ellos hayan realizado sus ejecuciones cuando el flujo de ejecución general de la aplicación retoma el control. El patrón paralelo debe estar compuesto por dos o más caminos de ejecución, donde cada uno de estos es llamado *rama de ejecución.* Véase la figura 2.3.

- **Condicional**. Un componente que implementa un patrón condicional es en el que solamente se ejecuta una de sus ramas de ejecución. Cada rama tiene una probabilidad p de ser ejecutada. Existirán n probabilidades, correspondiéndose

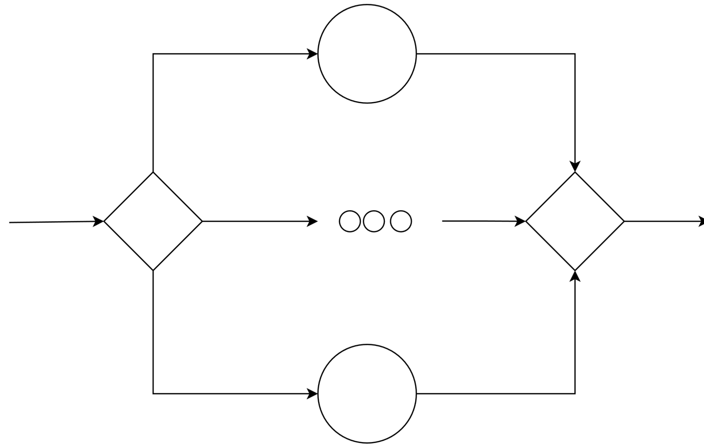


Figura 2.3: Patrón arquitectónico paralelo.

con las n ramas. La suma de estas probabilidades debe dar 1. El flujo de ejecución de la aplicación general se retomará cuando se haya terminado de ejecutar la rama elegida. Véase la figura 2.4.

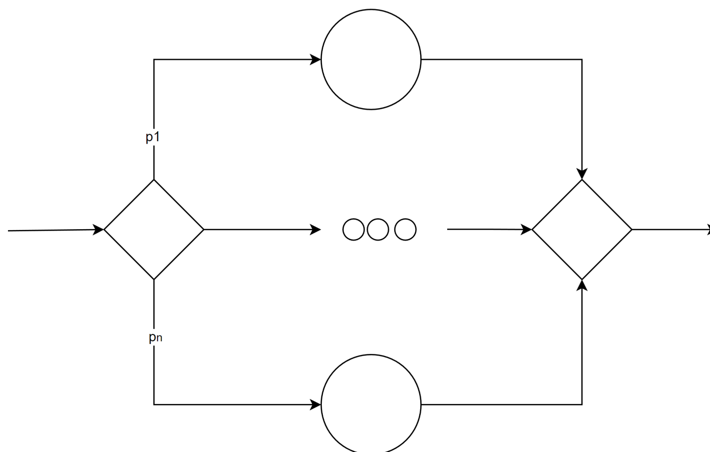


Figura 2.4: Patrón arquitectónico condicional.

- **Iterativo**. El patrón iterativo está compuesto de un nodo que deberá ejecutarse cierto número de veces y no es hasta que se haya ejecutado todas esas veces, cuando el flujo de ejecución de la aplicación general, se reanuda.

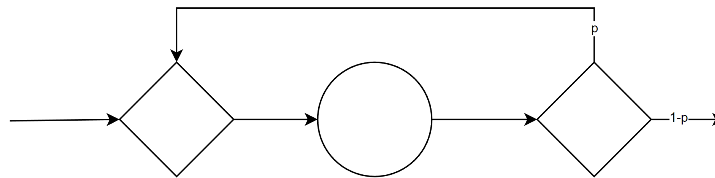


Figura 2.5: Patrón arquitectónico iterativo.

2.3. Atributos de calidad de servicio

Una vez conocidos los conceptos principales sobre la arquitectura de la aplicación, entra en consideración un factor principal y que juega un rol muy importante en el problema de la composición de servicios: los *atributos de calidad de servicio*.

Los atributos de calidad de servicio, a partir de ahora denotados como *atributos QoS* por simplicidad, son atributos que incumben al proveedor en sí. Los atributos que se van a trarar en este trabajo son: *coste, tiempo de respuesta, disponibilidad, fiabilidad, ancho de banda y latencia*. Cada autor que ha tratado este problema utiliza una variación de lo que entendemos por estos atributos debido al contexto en el que se puede aplicar su solución, por tanto, es necesario puntualizar la manera en la que los trataremos.

- **Coste**. En este trabajo nos centramos en el coste por uso, que no es más que la cantidad que deberá pagar el usuario para que su aplicación sea ejecutada. Algunos autores lo consideran como coste por despliegue, que es el coste asociado a lanzar la aplicación en el proveedor.
- **Tiempo de respuesta**. Es el tiempo necesario para que el proveedor ejecute el servicio.
- **Disponibilidad**. Es el tiempo efectivo que el proveedor está disponible para recibir y enviar peticiones. Se trata de un porcentaje, siendo 1 completa disponibilidad.

- **Fiabilidad**. Es la propiedad del proveedor que asegura que la respuesta que proporcione sea acertada y exacta. Se trata de un porcentaje. De igual manera que la disponibilidad, 1 denota máxima fiabilidad.
- **Latencia**. Es el tiempo requerido entre cada par de nodos para comunicarse.
- **Ancho de banda**. Se considera como la cantidad de información efectiva que se puede transmitir entre cada par de nodos.

Atentiendo al atributo a tratar, podemos discernir entre *atributos positivos* y *negativos*. Se considerará como atributo positivo aquel atributo que, mientras mayor es su valor, mejor será su calidad. Un ejemplo de esto es la *fiabilidad*, mientras más cercano al 100 % (1.0) mejor será su calidad. En cualquier otro caso, se considerará como atributo negativo.

Estos atributos QoS son los que nos permitirá discriminar las composiciones obtenidas. Gracias a la agregación de los valores de estos atributos, es decir, lo que valen a nivel global en la aplicación, podremos comparar composiciones distintas. Si solamente se tiene en cuenta un único atributo, la evaluación es trivial. Si el atributo involucrado es negativo, cuando se valoren dos composiciones distintas a y b, en el supuesto de que a obtenga un valor agregado menor que b. Se considerará a como mejor composición que b. Se aplicará el caso contrario para los atributos positivos.

Sin embargo, esto no resulta tan simple cuando se tratan varios atributos QoS a la vez. ¿Qué composición es mejor que otra? ¿Qué ocurre cuando para ciertos atributos una composición es mejor que otra y viceversa?. Para poder solventar esta cuestión se aplicará una función objetivo. Véase la fórmula 2.1. Es una función que nos proporcionará y dará como resultado la calidad de la composición en un rango discreto entre 0 y 1, siendo 0 la peor de las composiciones y 1 la mejor. En vista del usuario, no tiene por qué tener la misma importancia todos los atributos QoS. Para este, puede tener más repercusión en su decisión, por ejemplo, el coste antes que otros atributos. Para poder tratar esta cuestión, se aplica un peso asociado a la relevancia que deberá tener el atributo en la función objetivo, siendo un valor que se sitúa entre 0 y 1, siendo 0 ninguna importancia y 1 la máxima. Es importante

recalcar que la suma de todos los pesos de todos los atributos tenidos en cuenta debe ser de 1.

$$F.O_c = \sum_{i=0}^{|QoS|} A_i * W_i \tag{2.1}$$

Siendo c la composición dada, $|QoS|$ el número de atributos QoS tenidos en cuenta, A_i el valor agregado del atributo QoS i-ésimo y W_i el peso del atributo i-ésimo.

En vista de esta situación, se deberá plantear la cuestión: ¿todos los atributos se agregarán de la misma manera, sea la arquitectura que sea? Como es de esperar, la respuesta es no. La arquitectura juega un papel muy relevante y decisivo en el valor agregado asociado a un atributo.

Supongamos el atributo QoS de tiempo de respuesta y un ejemplo trivial de 10 servicios, todos ellos de manera secuencial. Se debe llegar a la conclusión de que el tiempo de respuesta agregado, asociado a esa composición, será la suma de los tiempos de ejecución de los proveedores elegidos de cada uno de los servicios, ya que el servicio precedesor no puede empezar su ejecución si el antecesor no ha finalizado la suya. En cambio, si nos encontramos con un patrón paralelo de 10 ramas, con un servivio en cada una de ellas, el tiempo de respuesta será limitado por la rama cuyo tiempo sea el mayor. Por esta misma razón, se han utilizado una serie de fórmulas de agregación para que el valor agregado respete la arquitectura subyacente a esta. Las fórmulas para coste, tiempo de respuesta, fiabilidad son las propuestas por Cardoso en [4]. La disponibilidad se trató como una adaptación de la fiabilidad. Los atributos de latencia y ancho de banda aplican nuestra propia definición. En la tabla 2.1 se puede ver las expresiones de estas fórmulas de agregación por patrón y atributo QoS.

Retomando la definición de componente ya descrita en la sección anterior, la aplicación de manera recursiva de estas fórmulas finalizará con un único valor llamado el *valor total agregado de la aplicación para cierto atributo QoS*.

El último concepto relacionado con los atributos QoS es el de *restricción*. Las restricciones son condiciones que la composición deberá cumplir. Un ejemplo podría

	Sequential	**Parallel**	**Conditional**	**Iterative**
Cost (-)	$\sum_{i=0}^{n} C_i$	$\sum_{i=0}^{n} C_i$	$\sum_{i=0}^{n} p_i C_i$	$C_i/(1-p_i)$
Response time (-)	$\sum_{i=0}^{n} T_i$	$max(T_i)$	$\sum_{i=0}^{n} p_i C_i$	$T_i/(1-p_i)$
Latency (-)	$\sum_{i=0}^{n} L_i$	$max(L_i)$	$\sum_{i=0}^{n} p_i L_i$	$L_i/(1-p_i)$
Availability (+)	$\prod_{i=0}^{n} A_i$	$\prod_{i=0}^{n} A_i$	$\sum_{i=0}^{n} p_i A_i$	$(1-p_i A_i)/(1-p_i A_i)$
Reliability (+)	$\prod_{i=0}^{n} R_i$	$\prod_{i=0}^{n} R_i$	$\sum_{i=0}^{n} p_i R_i$	$(1-p_i R_i)/(1-p_i R_i)$
Throughput (-)	$min(Tr_i)$	$min(Tr_i)$	$\sum_{i=0}^{n} p_i Tr_i$	$min(Tr_i)$

Tabla 2.1: Fórmulas de agregación aplicadas en este trabajo.

ser el de coste menor que 50 y de disponibilidad mayor que el 90 %. Para esto, el algoritmo intentará aproximarse lo máximo posible a los requerimientos del usuario, si que existe solución. Las restricciones no son obligatorias y en caso de no establecerlas, se buscará la mejor composición. Entiéndase como mejor composición, la que maximice la función objetivo.

2.4. Estado del arte

Como ya se mencionó en el capítulo de introducción. El problema de la composición de servicios es un tema que cada vez está tomando más relevancia gracias a su gran utilidad. Por esta misma razón, no son pocos los autores que han propuesto sus propios enfoques para poder tratar este problema. Cada uno de estos métodos sirven en unas determinadas circunstancias y algunos conllevan una serie de limitaciones en ciertas situaciones. Dentro de las diferentes técnicas y enfoques se pueden encontrar principalmente dos variantes, la primera de ellas son los métodos exactos y la segunda los métodos basados en heurísticos. Primero realizaremos unas pequeñas aclaraciones para entender el enfoque de los autores citados.

Los autores mencionados comienzan, como punto de partida, con una descripción de la aplicación o lo que nosotros llamamos: arquitectura. Esta descripción puede ser entendida como la definición de los nodos y la manera en la que interaccionan entre ellos, es decir, los patrones que contienen, el orden de estos y la colocación de los nodos. Según el trabajo que se estudie, se tendrán en cuenta más o menos

estructuras de control, es decir, los patrones arquitectónicos.

Los nodos son las entidades, que como mencionamos, son los que se encuentran interconectadas en cierto orden dentro de la aplicación. Estos nodos, sujetos al autor, pueden ser denotados de distintas formas, algunas de ellas son: *servicio abstracto, tarea* o símplemente *servicio*. Ellos, a su vez, tienen una serie de objetos de los cuales solamente pueden elegir uno. Estos elementos, para nosotros denotados como *proveedores*, pueden ser llamados como *servicios concretos* en ciertos trabajos. Los proveedores contienen, por otro lado, una serie de atributos, los ya explicados como atributos QoS. Atendiendo al trabajo, se tratarán más o menos número de atributos distintos. Esto depende principalmente del contexto en el que los autores esten resolviendo el problema. Estos atributos a su vez pueden ser entendidos de diferentes maneras por la misma razón. Un ejemplo de esto puede ser el coste. En nuestro trabajo lo entendemos como el coste por uso, pero otros autores, dado su enfoque, deberán obtener el coste por despliegue.

El punto en común de todos los trabajos tratados en esta sección, es la búsqueda de la mejor composición atendiendo a una situación concreta, bien buscan la mejor composición, o bien dan una composición sujeta a las restricciones únicamente, o ambas a la vez.

Uno de los métodos exactos que podemos encontrar es la que propone Zeng et al. en [10] en la que aplica programación entera. Esto no es más que un enfoque que busca todas las posibles composiciones que pueden encontrarse en la aplicación dada y finalmente, elegir la que tenga una mejor calidad de solución. Tal como resaltan dichos autores, este enfoque crece exponencialmente a medida que la entrada aumenta. El mayor beneficio que trae este método, es que se conoce con certeza que no existe una mejor composición que la dada como solución. Es un método ideal para espacios de búsqueda reducidos. Se trata de un método que busca de manera global la mejor composición.

Entrando dentro de las soluciones basadas en heurísticos, existen varios enfoques que no buscan la solución de manera global aplicando la función objetivo con los valores

agregados de la aplicación, sino que lo hacen de manera local. Vamos a presentar tres trabajos que siguen la filosofía de buscar de manera local mediante un *divide y vencerás*. El primero de ellos, es el que propone Pozas et al. en [6]. Los autores hacen uso de esta técnica y buscan, dentro del servicio, el proveedor que maximice la función objetivo propuesta por ellos. Como norma utilizan el divide y vencerás y en caso de encontrar un patrón arquitectónico paralelo, aplican un algoritmo genético. Posteriormente recomponen la solución. En su trabajo, señalan que empleando un divide y vencerás únicamente, no obtenían la solución más optima cuando encontraban un patrón paralelo. El uso de un divide y vencerás reduce drásticamente el tiempo de ejecución, ya que la complejidad que se planteaba como exponencial en primera instancia, se convertirá en orden lineal, excepto cuando localizan un patrón paralelo, en cuyo caso deben llamar al genético y dicha complejidad no será lineal, pero igualmente es drásticamente más veloz que aplicar un algoritmo genético a la aplicación completa. Uno de los problemas que trae este método consigo es que se pierde la posibilidad de definir restricciones sobre la aplicación. Yu y Lin, en [8] propusieron un enfoque en el que se pueden definir restricciones utilizando también divide y vencerás. El principal problema de esta técnica es que para poder aplicar y comprobar que las restricciones se han cumplido, será necesario realizar varias ejecuciones y por tanto, se perdería el principal beneficio que trae esta técnica consigo, su velocidad. Berner et al. en [2] defienden que, en el momento de mirar el problema de manera global, el uso de un divide y vencerás no siempre dará la mejor solución y que a medida que el número de servicios o nodos aumente la solución óptima se alejará más.

Uno de los enfoques más utilizados dentro de los heurísticos, es el del empleo de algoritmos genéticos. Aunque en capítulos posteriores se describirá con mayor detalle, un algoritmo genético consiste básicamente en buscar, de manera aleatoria y bajo ciertas condiciones, una aproximación a la solución. Uno de los primeros autores en usar esta técnica para la composición de servicios fueron Canfora et al. en [3]. Para los autores, la solución se codificará en una lista con tantas posiciones como servicios tenga la aplicación. Dentro de cada posición se guardará un número que será el

que indique el proveedor elegido para el servicio de esa posición. Véase figura 2.6. La generación de estas lista se realizará de manera aleatoria y en varias etapas. El proceso concluirá con una lista que al aplicarle una función objetivo, obtendrá una calidad superior que el resto de listas generadas. Esta será la composición solución.

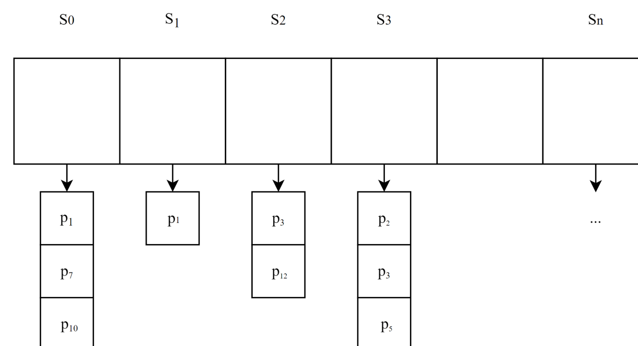


Figura 2.6: Representación de un individuo de la población

Los problemas que trae este método, al igual que todos los métodos no exactos, es que no se puede saber con certeza si la solución obtenida es la mejor o cómo de cerca de esta se encuentra. Así mismo, otra desventaja es que será necesario efectuar cálculos muy costosos dentro del mismo, aumentando su tiempo de ejecución.

Un último enfoque bastante extendido que se ha aplicado a este problema, es un método heurístico, y lo que nosotros en este trabajo denotamos como *el método basado en la utilidad*. Es un procedimiento en el que la cantidad de proveedores y la calidad de los valores de los atributos QoS que proporcionan dichos proveedores se tienen en consideración. Uno de los primeros autores en tratarlo de esta forma son Farhad et al. en [5]. Este procedimiento lo denotan como QCD. Se trata de un enfoque que evita precisamente el problema que trae consigo el uso de un algoritmo genético: cómputos muy costosos dentro de la llamada de este. El procedimiento de utilidad realiza un pre cómputo asignándole valores a los atributos de calidad de servicio dentro del servicio, de modo que, posteriormente en la llamada del genético, solamente se realicen consultas sobre los datos ya existentes, evitando cálculos innecesarios. El procedimiento se divide en tres fases. La primera de ella es la de descomponer los atributos de calidad de servicio por servicio y en intervalos, asignándoles a cada uno de estos un valor. Estos intervalos son los llamados *gra-*

dos de utilidad. La segunda fase es la llamada del algoritmo genético en la que se buscará la utilidad requerida por cada atributo QoS para satisfacer las restricciones impuestas. Finalmente, se llevará a cabo una conversión de la salida del genético a composición. Este método será explicado en mucho mayor detalle en la sección 4.3. Yuan et al. en [9], realizaron una pequeña mejora para obtener una mejor calidad en la solución obtenida. Añaden un mecanismo que calcula de manera automática un parámetro necesario en el método QCD de Farhad, los llamados *número de rangos grados de utilidad*. Lo efectúan mediante el uso de lógica difusa. En el procedimiento de Farhad et al. en [5] es necesario indicar el número de grados, es decir, el número de intervalos.

Este método trae consigo una principal ventaja: para entornos dinámicos, solamente bastará con recalcular ciertos datos del pre cómputo para volver a aplicar el algoritmo genético y evitar cálculos innecesarios. Además, soporta restricciones dadas por el usuario. Este procedimiento tiene una desventaja principal, y es que los atributos que dependen del canal, como la latencia o el ancho de banda, no pueden ser tratados porque, como se explicará posteriormente, los atributos asociados a los proveedores deben ser aquellos que dependan únicamente de ellos. Al tener en cuenta los canales de comunicación, el valor del atributo QoS dependerán de cada par de proveedores involucrados, no únicamente del propio proveedor elegido para el servicio. En este trabajo, hemos desarrollado un algoritmo basado en un nuevo heurístico para poder gestionarlos y poder proporcionar composiciones con atributos que dependan de los canales de comunicación.

Concluimos esta sección con tres tablas resumen en la que comparamos todos los métodos mencionados:

	Prog. entera	DyV	DyV varias pasadas
Tiempo de ejecución	Muy elevada	Desestimable	Elevada
Permite restricciones	Sí	No	Sí
Da mejor solución	Sí	No	No

	GA completo	Utilidad
Tiempo de ejecución	Elevada	Elevada
Permite restricciones	Sí	Sí
Da mejor solución	No	No

	Búsqueda local	Búsqueda global
Programación entera	No	Sí
Divide y Vencerás	Sí	No
DyV varias pasadas	Sí	No
GA completo	No	Sí
Utilidad	Sí	Sí

CAPÍTULO 3

Conocimientos previos

En este apartado se darán las primeras nociones y los conocimientos necesarios para que en los apartados siguientes se tenga el trasfondo para entender este método. Se realizará un breve resumen de en qué consiste los métodos basados en utilidad y los algoritmos genéticos.

3.1. Introducción al método de utilidad

Recordemos el esquema básico del servicio en sí. Cada servicio tiene una serie de proveedores disponibles. No necesariamente todos los servicios tienen los mismos proveedores ni tampoco el mismo número. Cada proveedor lleva asociado unos valores de los atributos de calidad de servicio. Para cada proveedor, cada uno de los valores de estos atributos puede diferir.

En nuestro trabajo proponemos dos enfoques distintos dependiendo del problema a resolver. A continuación se describirá cuando se puede aplicar cada uno de ellos.

- Nivel de utilidad del proveedor, por simplicidad denotado como *UR*.
    - Problemas sin restricciones.
- Heurístico + GA + elección del proveedor con el valor de utilidad más cercano. A partir de ahora llamado *GA utilidad* o *UGA*, por simplicidad.

- Problemas con restricciones.
- Problema sin restricciones.

El primer método consiste en aplicar una función objetivo a los proveedores del servicio. Es una alternativa al método basado en la utilidad que se nutre de los datos obtenidos de su pre cómputo para poder realizar los cálculos necesarios en el mismo. Este segundo enfoque aplica cierta función objetivo de manera local, por lo que no se podrán utilizar restricciones.

El método de GA-utilidad establece un balance entre el número de proveedores que tiene el servicio y la calidad de los valores que estos tienen. Una premisa en la que se basa este procedimiento es: *No sirve de nada un proveedor que destaque en un único valor si en el resto de sus valores son mediocres.* El procedimiento generará un número de rangos de tantos valores como el usuario indique, esto es el llamado *número de grados de utilidad*. La división de los rangos se realiza por atributo QoS y por servicio. Atendiendo al número de rangos en el que se divida cada atributo, se podrá buscar con mayor granularidad el grado óptimo para dicho atributo. Entonces, ¿por qué no se divide en un número de rangos muy grande?. La primera razón es por una mera cuestión de escalabilidad y la segunda, y más importante razón, es que al dividirlo por un número de rangos tan elevado, los datos se verán deformados y esto acarreará que no se llegue a una solución óptima. Esto concluirá con una lista en cada servicio de rangos por cada atributo QoS. Esta cuestión será explicada con mayor detalle en la sección 4.3.

La idea subyacente, y lo que se busca realmente, es el de realizar unos rangos divididos por atributo de calidad de servicio en los que se puedan tener en cuenta el número de proveedores del servicio que pueden ofrecer dicho valor y que también figure cuan bueno es dicho atributo. Esto permitirá llevar a cabo una división de los atributos a nivel local, es decir, a nivel del proveedor para que, posteriormente, se pueda realizar una búsqueda por atributo de calidad de servicio a nivel global. Gracias a la división de cada proveedor en rangos de utilidad por atributo, será fácilmente comprobable que toda la aplicación cumpla la utilidad requerida por las

restricciones de cada atributo QoS impuestas.

Teniendo esos datos, se realiza la llamada al algoritmo genético en donde las restricciones del usuario se normalizan entre el máximo y mínimo agregado de la aplicación. A cada atributo QoS de la restricción se le asignará un valor entre 0 y 1. La llamada del algoritmo genético buscará los valores de los índices de los grados de utilidad que mayor valor otorgue. La salida del genético será una lista en la que se puede saber cuál es el grado óptimo para cada servicio y cada atributo QoS. Se deberá efectuar una traducción de esta solución a composición. De nuevo, los detalles más técnicos serán explicados en la sección 4.3.

El método de GA-utilidad o UGA será el método de preferencia cuando se busque una composición que respete las restricciones. En cambio, cuando no se tengan en cuenta, la solución propuesta por el primer método (Nivel de utilidad del proveedor o UR), será la elegida por sus grandes beneficios y solución más que suficientemente aceptable.

3.2. Adaptaciones necesarias

En la sección anterior se explicó a grandes rangos el método basado en la utilidad. Este método se basa en la información proporcionada por los otros proveedores candidatos del servicio para obtener los rangos y aplicar el proceso completo. Sin embargo, ¿qué ocurre sí tratamos atributos que dependen de cada par de proveedores?

Uno de los problemas del método basado en la utilidad es que no puede tratar atributos dependientes del canal. Entiéndase por atributos dependientes del canal a los que dependen de cada par de servicios (o par de proveedores elegidos para esos dos servivios). Tal como se explicó, el método de utilidad realiza un baremo de los valores que se pueden obtener dentro del propio servicio. Para atributos como coste, tiempo de respuesta, disponibilidad y fiabilidad, no existe ninguna problemática, ya que estos atributos dependen únicamente del propio proveedor. Pero, dado un

escenario en el que se tiene en cuenta un atributo dependiente del canal, lo más común es que se plantee la pregunta de: ¿Qué valor tendrá el proveedor para ese atributo si depende de la selección que haga y no llevo a cabo ninguna selección hasta finalizar el método de utilidad? Para ello, se ha realizado un heurístico que pretende simular el propio método de utilidad, es decir, asociarle un valor a priori que nos pueda dar nociones de la calidad del proveedor. Para este fin, hacemos una extrapolación del valor del canal a dicho servicio, asociando una aproximación de la latencia que se obtendría si se eligiera el proveedor del servicio. La *latencia* se calcula mediante la distancia que existe entre cada par de proveedores, a mayor distancia, mayor latencia. Por esto, se calcula la media de las latencias con cada proveedor candidato con cada uno de los proveedores del servicio siguiente. Véase figura 3.1. Esto nos dará una media de la latencia que se obtendría si se eligiera ese proveedor. Este valor es el que le daremos como la *latencia estimada para ese proveedor*.

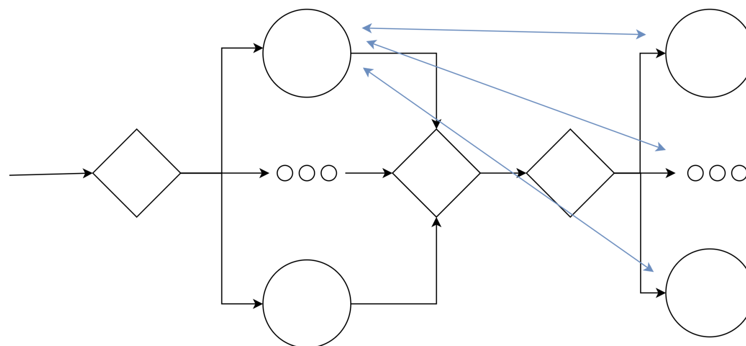


Figura 3.1: Representación del cálculo de las latencias con los siguientes

La aplicación consecutiva de este procedimiento nos dará, para cada uno de los proveedores de todos los servicios, sus latencias estimadas.

La lógica subyacente a este método es muy simple. A menor valor de la latencia supuesta para ese proveedor, será más probable que si se eligiera dicho proveedor en la composición, la latencia sea la óptima. Esto es debido a que la media saldrá mucho menor, mientras más cercano esté a los proveedores de las localizaciones de los siguientes. Por otra parte, una latencia estimada muy alta nos dará indicios de que no importa lo que se elija en sus siguientes, ya que lo elegido dará, muy

posiblemente, una composición peor respecto a la latencia que otro proveedor con una media menor.

Otro atributo que depende del canal es el de *ancho de banda*. En este trabajo hemos realizado unas simplificaciones para poder tratar este atributo. Tal como definimos, el ancho de banda es la cantidad de información efectiva que puede transmitirse entre cada par de nodos. Dados dos proveedores a y b, siendo a el antecesor de b, dado el caso de que a tuviera un ancho de 100 y b uno de 1000, no existiría ningún problema. En cambio, si fuera a el que pudiera transmitir 1000 y b pudiera recibir solamente 100, existiría 900 unidades que se perderían o que no podrían llegar. Este atributo, al igual que la latencia, depende de las selecciones que realicemos es decir, que es dependiente del canal y, por tanto, se realizará una simplificación para poder tratarlo. La simplificación que proponemos es el de suponer que el proveedor del servicio dará el máximo del ancho de banda y que no se perderá información. El usuario final será el encargado de poner los mecanismos necesarios para que dicha información no se pierda, con soluciones tales como un buffer o una memoria.

La última cuestión a tener en cuenta cuando hablamos de canales de comunicación, es que la comunicación debe efectuarse a través de un medio concreto. Este medio es por donde se manda el mensaje o la información entre cada par de nodos. Ejemplos de medios pueden ser el uso de bluetooth, wifi o fibra. Todos los nodos no tienen por qué tener entrada y salida del mismo tipo. Es sencillo pensar en el caso concreto de una mota, esta únicamente tendrá conexión mediante wifi o bluetooth. Un servidor, por el contrario, lo más probable es que únicamente tenga conexión mediante fibra.

En nuestra propuesta nos abstraemos del canal de comunicación concreto a utilizar. Esto no tiene por qué ser un impedimento para el empleo de nuestra solución, pues siempre podemos suponer que se introducen los mecanismos necesarios para que la conexión pueda llevarse a cabo, por ejemplo, con switches, routers, etc. La alternativa más general sería considerar los canales al mismo nivel que los servicios, con unas opciones similares a la que ofrecen proveedores para servicios. Sin embargo, esto incrementaría considerablemente la complejidad del problema, y dejamos como trabajo futuro su posible utilización.

3.3. Algoritmos genéticos

En esta sección se explicará con mayor grado de detalle lo que es un algoritmo genético y el contexto del problema, así como una serie de definiciones básicas a las que posteriormente se hará referencia.

Los algoritmos genéticos son una de las técnicas más ampliamente utilizadas en la composición de servicios y muchos autores lo tomaron como el mejor método para poder resolver este problema. Como se puede ver en [1], un algoritmo genético es una técnica que se inspira en la propia evolución. La solución será generada de manera aleatoria dentro de una población inicial e irá evolucionando a medida que se generan más poblaciones. Solo aquellos individuos con mejor calidad de solución serán los que tengan descendencia y, por tanto, se irá refinando la solución hasta que finalmente converja en la final. Existen diferentes métodos para determinar cuándo debe converger la población. Los detalles y parámetros de cómo aplicamos el algoritmo genético serán explicados posteriormente. Los algoritmos genéticos están basados en la propia aleatoriedad y, por tanto, pueden existir casos en los que sea difícil llegar a una solución que sea lo suficientemente buena en un tiempo razonable. Por esta razón, es vital elegir los mejores parámetros atendiendo a cómo se trate el problema.

Lo primero que tenemos que tener en cuenta es que tendremos que tener un mecanismo para poder codificar cada uno de los individuos de la población. Cada uno de estos individuos será una posible solución al problema. En la figura, la 3.2, se puede ver las diferentes partes del genotipo. El genotipo no es más que la manera en la que se codifica el cromosoma y este es la estructura donde se almacenarán las soluciones en su debido formato.

Cada miembro de la población será representado con una serie de genes que deberán ser del tipo de dato (String, Integer, ...) que requiera el problema. En nuestro problema concreto, serán enteros que irán desde el 0 al número de grados de utilidad definidos. El conjunto de estos genes en un orden concreto será un miembro de la

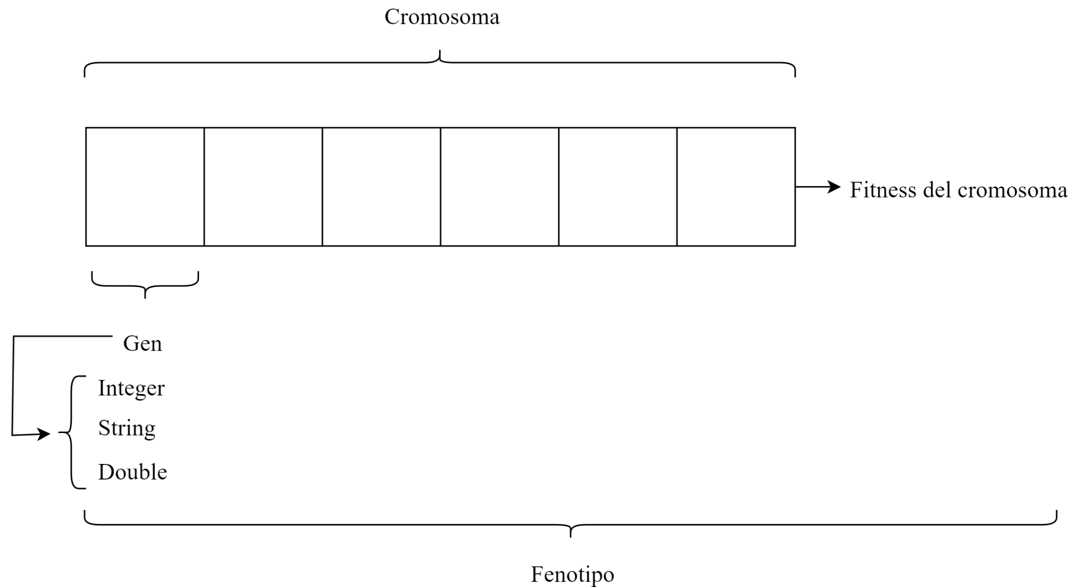


Figura 3.2: Representación de un genotipo genético

población. También estos valores en su correspondiente lugar o también llamado *alelo*, será llamado *cromosoma*.

Una vez definida la manera en la que se representa y se genera la primera población, será necesario un mecanismo para poder saber el grado de adaptación al problema de cada uno de estos individuos. Para esto, tenemos una función llamada *fitness*, que nos dirá la calidad de la solución de cada uno de ellos. A mayor fitness, mejor calidad de solución será.

Hecho esto, se pasará a la fase reproductiva, en la que los mejores individuos (con mejor fitness), serán los que se combinen para producir nuevos individuos. Es más probable que, eligiendo a los mejores individuos, los nuevos generados sean de una calidad similar o superior. Seleccionados los padres, será necesario realizar los operadores de cruce y mutación. En nuestro trabajo no hemos efectuado la operación de cruce debido a que no tiene sentido en el contexto de los métodos basados en la utilidad.

En cambio, en nuestro trabajo sí que empleamos el segundo operador, el de mutación. Este operador se aplica a cada hijo. Simplemente, uno de los genes cambia a otro aleatorio de su lista de genes disponibles. En la figura 3.3 se puede ver cómo afecta

al cromosoma en particular.

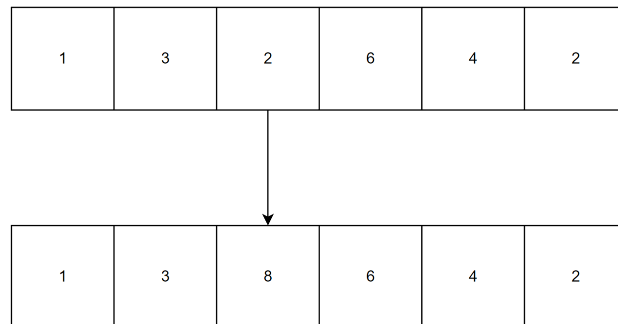


Figura 3.3: Mutación de un cromosoma

La aplicación de un algoritmo genético puede resumirse en:

1. Inicializar la población
2. Evaluar la población
3. Obtener los mejores individuos
4. Cruzar los mejores individuos (no en nuestra propuesta)
5. Mutar los hijos
6. Volver al punto 2.

¿Hasta cuándo será necesario repetir este proceso? Para poder finalizar la aplicación del genético se inventó un mecanismo llamado *criterio de parada*, que no es más que una condición que cuando se cumpla parará el algoritmo, obteniendo como solución el cromosoma con mayor fitness. Algunos ejemplos de criterios de paradas son los que se enumeran a continuación:

- **Parada por tiempo**: Se finaliza la generación pasado el tiempo que se le indicó.
- **Convergencia de la población**: Se finaliza la generación si el fitness de la población no difiere en un cierto porcentaje establecido.

- **Fitness umbral**: Finaliza la generación si se obtiene el fitness umbral establecido por el usuario. Si se desea minimizar, la población deberá estar por debajo del mismo, el caso contrario se aplicará cuando se quiere maximizar.

Nuestra solución está implementada en el lenguaje de programación *Java* por esta razón, la librería que hemos utilizado para implementar el algoritmo genético es *Jenetics* [7]. Es una librería muy eficaz y que puede ser utilizada para resolver diversos problemas de optimización. Las clases más importantes que implementa esta librería son las siguientes:

- Gen (gene): contiene la información de la codificación de la solución. Todos los genes son inmutables.
- Cromosoma (Chromosome): clase que contiene una colección de genes y que al menos debe tener uno. En nuestro caso emplearemos IntegerChromosome debido a las necesidades del problema.
- Genotipo (Genotype): es la clase central y con el que trabaja el mecanismo de evolución.
- Fenotipo: consiste en la representación de un individuo. Está compuesto por un genotipo y su valor de fitness.

La función de fitness aplicada en nuestro problema será descrita en la sección 4.3.

Finalmente, el criterio de parada aplicado en nuestra solución es el *Steady Fitness*. El proceso de evolución finalizará si el mejor fitness no ha cambiado en cierto número de generaciones.

CAPÍTULO 4

Métodos basados en la utilidad

Este capítulo se explicarán las soluciones propuestas junto con los escenarios en los que se deberá aplicar cada uno de los procedimiento de utilidad introducidos previamente.

4.1. El método de utilidad

Después de haber acabado el preámbulo de los conocimientos necesarios para poder entender el método, en esta sección se describirán el procedimiento, los cálculos necesarios y la manera en la que se tratan los datos. Ya se adelantó en secciones anteriores, que disponemos de dos enfoques distintos, uno que buscará la mejor composición sin tener en cuenta restricciones y otro que sí que las tiene en cuenta. En la sección 4.2 se verá en más nivel de detalle el procedimiento a ser aplicado sin restricciones y en la sección 4.3 el procedimiento que debe ser aplicado teniendo restricciones.

De los dos métodos presentados en este trabajo, el que no tiene en cuenta las restricciones se resuelve en una única fase, tal como se verá en la sección 4.2. En cambio, el procedimiento que sí que las tiene en cuenta se divide en las tres siguientes fases.

- **Fase 1: Pre-cómputo.** En esta fase se calcularán los datos necesarios para que, en la llamada del algoritmo genético, solamente se tengan que realizar

cálculos menores y consultas a los datos ya calculados. La principal razón de este paso, es la de evitar cálculos costosos y repetidos durante la ejecución del genético. Este paso divide localmente los atributos tenidos en cuenta del propio servicio en una serie de rangos. No se utiliza en ningún momento datos del estado o disposición de la arquitectura.

- **Fase 2: Aplicación del algoritmo genético**. Se procederá con la llamada del algoritmo genético obteniendo como resultado un cromosoma con la utilidad óptima para ese problema. Esta búsqueda se efectúa de manera global, es decir, respecto a la arquitectura dada.
- **Fase 3: Conversión cromosoma - composición**. El cromosoma resultante se tiene que decodificar, ya que dará como resultado una secuencia de índices. El resultado después de esta fase será la de la composición final.

4.2. El método de utilidad sin restricciones

El procedimiento realizado es muy parecido a una aplicación de un divide y vencerás clásico. Dividir el problema hasta llegar a la unidad indivisible, efectuar el cálculo necesario y posteriormente recomponer la solución. El tiempo para este método es lineal. En la sección de experimentos se verá más información de dicho tiempo de ejecución, así como los resultados que se obtienen del mismo.

Este método consiste en asociar un valor de cuan bueno es un proveedor dentro de un servicio, asociándoles valores entre 0 y 1. A mayor valor, mejor proveedor es.

Para esto, se buscará para cada uno de los atributos QoS y cada servicio, el menor y mayor valor (de un atributo concreto) que se puede obtener dentro de los proveedores candidatos del servicio. Para cada uno de los atributos QoS del proveedor, se normaliza el valor respecto a ese mínimo y máximo. Véanse las ecuaciones 4.1 y 4.2 utilizadas para normalizar atributos negativos y positivos respectivamente. Donde q_k^{min} y q_k^{max} son los valores mínimo y máximo del atributo QoS k dentro del servicio

y $q_k^{proveedor}$ el valor para ese atributo de un proveedor concreto.

Finalmente, sumando todos los datos normalizados por el peso de cada atributo QoS, se obtiene la calidad del proveedor. Véase la ecuación 4.3, donde se obtiene la utilidad del proveedor i, del servicio j para el atributo QoS k. El valor resultante de UT_i^j será la calidad del proveedor. Dese cuenta de que no necesariamente el mismo proveedor debe tener el mismo valor de utilidad, ya que dicho valor depende únicamente de los otros proveedores candidatos del servicio.

$$norm_k = \frac{q_k^{max} - q_k^{proveedor}}{q_k^{max} - q_k^{min}}, \text{si k negativo.} \tag{4.1}$$

$$norm_k = \frac{q_k^{proveedor} - q_k^{min}}{q_k^{max} - q_k^{min}}, \text{si k positivo.} \tag{4.2}$$

$$UT_i^j = \sum_{i=0}^{|QoS|} norm_k * W_k \tag{4.3}$$

Consideremos el ejemplo de la figura 4.1. Se trata de un servicio que dispone de tres proveedores. Los atributos tenidos en cuenta son coste, tiempo de respuesta y disponibilidad. Coste y tiempo de respuesta son atributos negativos y disponibilidad es positivo.

Supongamos que cada uno de los atributos QoS tienen la misma importancia, por tanto, los pesos asociados a ellos será de 1/3. Aplicando las respectivas fórmulas correspondientes obtendremos los datos mostrados en la tabla 4.1. Dados los valores de utilidad obtenidos, concluimos con que el mejor proveedor es el *proveedor 2*.

	Coste	Tiemp. Resp.	Disponibil.	Utilidad
Proveedor 1	30	50	0.99	0.5277
Proveedor 2	20	20	0.95	0.6667
Proveedor 3	35	60	0.96	0.0833

Tabla 4.1: Resultados de aplicar la normalización y los pesos a los proveedores

La aplicación de este procedimiento para cada uno de los servicios de la aplicación o

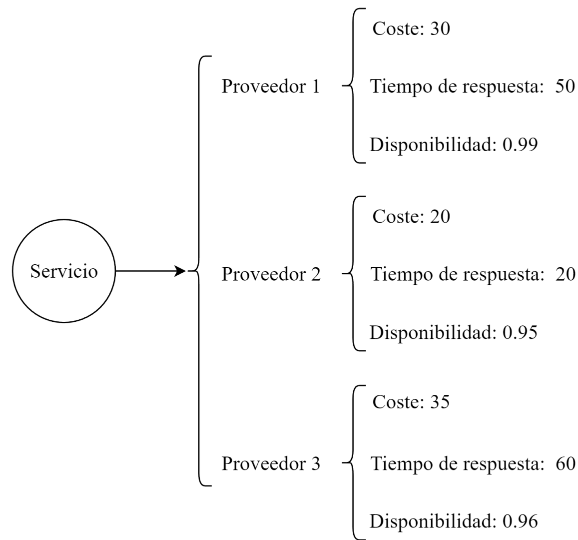


Figura 4.1: Ejemplo de un servicio con tres proveedores disponibles

arquitectura dada finalizará con una composición pseudo-óptima, eligiendo el mejor proveedor para cada servicio.

Como se puede apreciar, este método es extremadamente rápido en términos de computación. Su complejidad algorítmica se presenta como lineal. También, se puede observar que no es posible aplicar restricciones, ya que, al tratarlo individualmente, a nivel del propio servicio, será imposible aplicarlas porque no disponemos de ninguna clase de valor que nos indique el valor mínimo que debe valer la utilidad de los proveedores o de qué valor no se debe exceder. Por tanto, el siguiente método descrito en la sección 4.3, será utilizado cuando se tengan restricciones.

4.3. El método basado en la utilidad con restricciones

En este apartado se presentan los pasos y cálculos necesarios para realizar el procedimiento con restricciones. Es importante tener en cuenta que también se puede adaptar para buscar la mejor composición sin restricciones, pero tal como se verá

en el apartado de *experimentos*, el método de la sección 4.2 dará mejor resultado en la mayoría de los casos.

4.3.1. Primera fase: pre-cómputo

La utilidad comienza con un pre-cómputo que se almacenarán en unas estructuras de datos para que, posteriormente, el algoritmo genético solamente realice consultas. Con esto, se ahorrará cómputo innecesario. Esta fase, a su vez, está dividida en una serie de pasos.

Obtención del mínimo y máximo por servicio

El primer paso a seguir es el de obtener el valor mínimo y máximo de cada atributo QoS obtenido por los proveedores de cada servicio. Finalizará con una correspondencia: Servicio - Atributo QoS - (valor mínimo, valor máximo).

Matriz de calidad de servicio

Cuando ya se tienen los valores mínimos y máximos por servicio y atributo, se generará la *matriz de calidad de servicio*. Será una estructura que, por cada servicio y atributo QoS, se realiza una división de los datos en intervalos discretos, que en función de los grados de utilidad que se deseen aplicar, serán más o menos. Para clarificar la manera en la que se realiza, tomaremos el ejemplo de la sección anterior de la figura 4.1 y supondremos que queremos dividirlo en 4 grados de utilidad. La estructura quedará como el tabla 4.2.

Tomemos el atributo QoS de Coste. Observando la figura 4.1 se puede ver que para este atributo el mínimo que se puede obtener es de 20 y el máximo de 35. Aplicando la ecuación 4.4 se obtendrá el valor en el que se deben ir incrementando los valores desde el mínimo para llegar al máximo en los grados establecidos. $\frac{35-20}{3} = 5$ quedando 20, 20+5, 20+10 y 20+15 respectivamente.

$$\Delta = \frac{q_{i,k}^{max} - q_{i,k}^{min}}{GradUt - 1} \tag{4.4}$$

donde GradUt es el número de grados de utilidad definidos $q_{i,k}^{max}$ es el máximo valor

que se puede obtener dentro del servicio i para el atributo Qos k.

		Grado 1	**Grado 2**	**Grado 3**	**Grado 4**
Servicio 1	**Coste**	20	25	30	35
	Tiem. respuesta	20	33.3333	46.6666	60
	Disponibilidad	0.96	0.97	0.98	0.99
..	...	...	...	...	...
Servicio n	**Coste**				
	Tiem. respuesta				
	Disponibilidad				

Tabla 4.2: Matriz de calidad de servicio

Matriz de probabilidad

Teniendo los datos de la matriz anterior, se calcula la *matriz de probabilidad*, que no es más que una estructura que almacena la cantidad de proveedores del servicio que son capaces de llegar a cierto grado. Para el ejemplo anterior tendremos la siguiente estructura (tabla 4.3).

		Grado 1	**Grado 2**	**Grado 3**	**Grado 4**
Servicio	**Coste**	1/3	1/3	2/3	3/3
	Tiem. respuesta	1/3	1/3	1/3	3/3
	Disponibilidad	3/3	2/3	2/3	1/3
..	...	...	...	...	...
Servicio 1 n	**Coste**				
	Tiem. respuesta				
	Disponibilidad				

Tabla 4.3: Matriz de probabilidad

Se puede apreciar cómo la disposición de disponibilidad es distinta a la de coste y tiempo de respuesta. Esto se debe a que al ser un atributo negativo, la pregunta cambia de: ¿cuántos son capaces de llegar a ese valor? a ¿cuántos superan ese valor? Tomando coste y el grado de utilidad 1 (coste 20) y grado de utilidad 2 (coste 25), solamente llega a ese valor el proveedor 2 (con coste 20), para el grado de utilidad 3 (coste 30), llega el proveedor 1 (con coste 30) y proveedor 2 (con coste 20) y para

el grado de utilidad 4 (coste 35), todos los proveedores son capaces de llegar a dicho valor.

Matriz de utilidad normalizada

Es la última estructura y sobre la que se realizarán las consultas en el algoritmo genético. Se obtiene de multiplicar la *matriz de probabilidad* con la *matriz de calidad de servicio* normalizada. Un aspecto importante a la hora de hacer este cálculo es que será necesario introducir cierto *offset* para que a la hora de normalizar esta matriz, evitemos que el menor de sus valores no valga 0, y que posteriormente al multiplicar ambas matrices el valor sea 0. Este *offset* viene descrito por [5] y lo denota como *delta*.

El resultado de la matriz de utilidad normalizada será para el ejemplo de la figura 4.1 se muestra en el tabla 4.4:

		Grado 1	**Grado 2**	**Grado 3**	**Grado 4**
Servicio	**Coste**	1/12	1/6	1/2	1
	Tiem.respuesta	1/12	1/6	1/4	1
	Disponibilidad	1/3	1/2	1/3	1/3
..	...	...	...	...	...
Servicio 1 n	**Coste**				
	Tiem.respuesta				
	Disponibilidad				

Tabla 4.4: Matriz de utilidad normalizada

Una vez se tiene estos valores, será necesario obtener el grado de utilidad requerida por las restricciones impuestas. Para ello, se utilizará las ecuaciones 4.1 y 4.2, sobre el valor mínimo y máximo agregado que se puede obtener de la arquitectura dada calculada mediante las fórmulas de la tabla 2.1. Posteriormente, se normalizarán los valores de cada restricción de cada atributo QoS.

4.3.2. Segunda fase: Aplicación del algoritmo genético

Hechos todos los cálculos necesarios. Se utilizará un algoritmo genético para encontrar la solución pseudo-óptima. Tal como se explicó en las secciones anteriores, el algoritmo necesita de un genotipo (manera en la que se codifica en cromosoma). En

nuestro caso particular, será un genotipo de longitud *número-de-servicios* multiplicado por el *número de atributos QoS*. Cada gen se corresponderá con un índice de la tabla de utilidad normalizada.

En la siguiente figura 4.2 se puede ver cómo resultaría la codificación para el ejemplo mostrado de la figura 4.1.

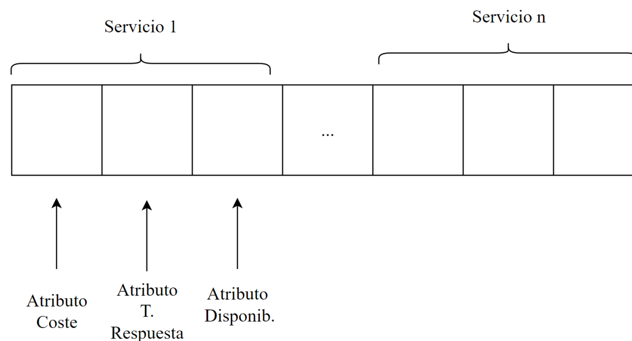


Figura 4.2: Genotipo de nuestro problema.

En nuestro caso particular, el gen será un número entero desde 0 hasta el número de grados de utilidad definidos menos uno.

Para poder discernir la calidad de la solución se aplica la función de fitness. Para cada uno de los índices para el atributo de calidad de servicio j, se obtienen los índices de su gen respectivo: $G_{i,j}$, es el gen para el atributo j y el servicio i. Se suman todos ellos y posteriormente se divide por el número de servicios. Véase ecuación 4.6. Este valor es comparado con la utilidad requerida por la restricción, y si la supera no se le aplicará ninguna penalización. En cambio, si no llega a dicho valor, se le aplicará una penalización como se indica en la ecuación 4.7, donde W_i es el peso asociado al atributo de calidad de servicio, f valdrá 0 si la restricción es cumplida y 1 si no lo cumple. P es el tamaño de la población.

$$\begin{aligned} &\text{si el cromosoma es factible} \rightarrow fitness = \sum_{j=0}^{||QoS||} Ut_j \\ &\text{si el cromosoma no es factible} \rightarrow fitness = \sum_{j=0}^{||QoS||} Ut_j - Penalizaci\acute{o}n_G \end{aligned} \tag{4.5}$$

$$Ut_j = \frac{\sum_{i=0}^{||s||} G_{i,j}}{||s||} \tag{4.6}$$

$$Penalizaci\acute{o}n_G = (\sum_{i=1}^{||QoS||} W_i * f) + (1/P) \tag{4.7}$$

En nuestro trabajo, el criterio de parada es el de *Steadyfitness*, que significa que el fitness del mejor individuo no cambie en cierto número de generaciones.

4.3.3. Tercera fase: Conversión cromosoma - composición

Una vez que converja el algoritmo genético, este devolverá como resultado un individuo solución de la forma mostrada en la figura 4.2, en el que a partir de este genoma se obtendrá la composición.

La manera de interpretar la salida es viéndolo por servicio. Para el servicio 1, será necesario obtener los valores desde el gen 0 hasta el número de QoS - 1. Todos estos índices deberán ser sumados y posteriormente divididos por el número de QoS. Esto dará como resultado lo que llamamos *utilidad requerida por el servicio*, por tanto, será necesario asignarle el proveedor que más se aproxime a este valor concreto de utilidad porque como se apunta en [5], será más probable que se cumpla la restricción cuanto más se aproxime a este valor.

Los valores de la utilidad de los proveedores siguen el mismo procedimiento que el descrito en la sección 4.3.

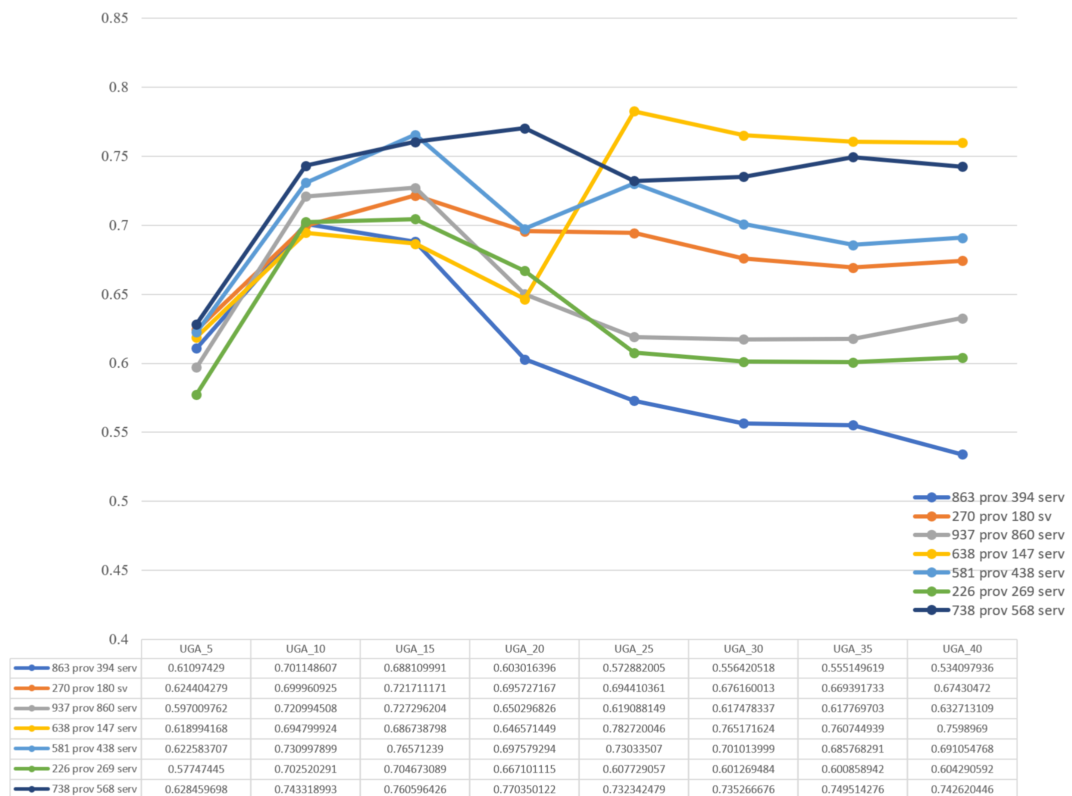
4.4. Grados de utilidad óptimos

El tener más o menos grados de utilidad puede influir en la calidad de la solución dada. Para saber cuál es el número de grados de utilidad que conduce a mejores soluciones, será necesario realizar una serie de experimentos para saber el número de grados exactos en el que dividirlos para maximizar la calidad de la solución dada. Debe ser un número en el que los datos o bien no se deformen por tener demasiados rangos o bien no sean suficientes y llegar a la granularidad que se requiere, obteniendo soluciones insuficientemente sesgadas. Los grados de utilidad son rangos en los que intervienen dos factores, el número de proveedores que llegan a un valor y lo bueno que es dicho valor.

A mayor número de grados uno puede intuir que se conseguirá una granularidad mayor, sin embargo, como se verá en los experimentos ejecutados, a mayor número de grados, peor fitness en la mayoría de los casos. Necesitamos tener la suficiente diferencia entre un grado y otro para poder establecer una clara división "mejor-peor grado-proveedor". En el momento que se tenga un número desproporcionado de rangos, podrán darse casos en los que la diferencia de un rango a otro sea de pocas décimas. Esto concluirá con una tabla de utilidad normalizada con diversos rangos bajo el mismo valor. En el caso contrario de tener demasiados pocos rangos, es muy probable que se dé el caso en el que grado elegido abarque diversas soluciones y no sea posible decantarse por la más acertada.

Se han realizado diferentes ejecuciones con diversos números de servicios y proveedores. Para cada número de servicios y proveedores se han llevado a cabo varias ejecuciones para obtener la media de la calidad de la solución. El principal motivo de esta decisión es que, tal como se dijo en la sección donde se explicó el funcionamiento del algoritmo genético, este algoritmo tiene el problema de que puede caer en mínimos y, por tanto, la calidad puede decrecer. Al realizar la misma prueba varias veces y obtener la media, podemos tener datos más certeros y precisos de su funcionamiento con esos parámetros.

En la figura 4.3 se pueden ver diferentes ejecuciones. Cada una de las líneas representa la media de las ejecuciones de la misma aplicación para cada uno de los números de grados. Cada ejecución varía de manera aleatoria el número de proveedores y servicios para ver cómo se comporta ante diferentes situaciones. UGA_x denota la resolución mediante el método de utilidad con algoritmo genético descrito en la sección 4.3. El eje x el número de grados de utilidad utilizados y el eje y, la calidad de la solución. Siendo 1 la de mayor calidad.



	UGA_5	UGA_10	UGA_15	UGA_20	UGA_25	UGA_30	UGA_35	UGA_40
863 prov 394 serv	0.61097429	0.701148607	0.688109991	0.603016396	0.572882005	0.556420518	0.555149619	0.534097936
270 prov 180 sv	0.624404279	0.699960925	0.721711171	0.695727167	0.694410361	0.676160013	0.669391733	0.67430472
937 prov 860 serv	0.597009762	0.720994508	0.727296204	0.650296826	0.619088149	0.617478337	0.617769703	0.632713109
638 prov 147 serv	0.618994168	0.694799924	0.686738798	0.646571449	0.782720046	0.765171624	0.760744939	0.7598969
581 prov 438 serv	0.622583707	0.730997899	0.76571239	0.697579294	0.73033507	0.701013999	0.685768291	0.691054768
226 prov 269 serv	0.57747445	0.702520291	0.704673089	0.667101115	0.607729057	0.601269484	0.600858942	0.604290592
738 prov 568 serv	0.628459698	0.743318993	0.760596426	0.770350122	0.732342479	0.735266676	0.749514276	0.742620446

Figura 4.3: Diferentes ejecuciones y parámetros para UGA

Podemos concluir que el número de grados óptimo para la mayoría de los problemas es de alrededor de 15. Si bien es cierto que existe alguna ejecución cuyo número óptimo de rangos no es 15, para la mayoría de estas sí que lo es. Es importante recalcar que este trabajo no se centra en resolver cada aplicación con su número óptimo, sino el de poder resolverlas con cualquier atributo de calidad de servicio, especialmente con los dependientes del canal, por esta misma razón se realizará una generalización y se utilizará a partir de ahora en las soluciones mostradas el 15 como grado de utilidad.

CAPÍTULO 5

Resumen de la implementación

En este capítulo se describirán las tecnologías aplicadas así como la descripción de las clases principales que realizan las funcinalidades más importantes del problema ya presentado en el capítulo 4.4. Este trabajo fue desarrollado dentro del grupo de investigación de F. Durán como parte del proyecto Sófocles.

- **Desarrollo**: el algoritmo fue desarrollado íntegramente en Java y la versión utilizada para las pruebas fue la 1.18.
- **Control de versiones**: se ha utilizado *github* para subir los cambios realizados. Como proyecto de investigación, se podía dar el caso de que cierta implementación que pretendía ser beneficiosa no lo fuera, por lo que esta herramienta nos permitía llevar un control de las versiones estables.
- **Entorno de desarrollo**: durante el proyecto se ha utilizado *Intellij ultimate.*
- **Librería**: Se ha utilizado la librería *Jenetics* para hacer uso de las funcionalidades que proporciona y poder implementar el algoritmo genético.

Las clases principales para utilizadas para la implementación del método basado en utilidad son las siguientes:

- **Service**: Clase que modela la entidad del *servicio* descrito en este trabajo. Contendrá una serie de candidatos que serán los proveedores disponibles para este.

- **Provider**: Clase que modela los *proveedores*. Tendrá atributos tales como la localización, un nombre y los valores de sus atributos QoS.

- **Una clase por cada patrón**: existe la clase *secuencial*, *iterativo*, *condicional* y *paralelo*. Cada una de ellas modela la manera en la que se comporta cada uno de los patrones. A su vez, estos patrones están dentro de una arquitectura y finalmente englobados en un componente. Modela la definición de componente descrita en la sección 2.3.

- **Application**: se trata de la clase que implementa las operaciones necesarias para tratar la arquitectura, componentes y servicios. Algunas de los datos que se hospedan en esta clase son:

  - Una lista con todos los proveedores de la arquitectura dada.
  - Una lista con todos los servicios de la aplcación.
  - Contiene las restricciones impuestas por el usuario.

  Las operaciones más importantes y utilizadas en este trabajo son:

  - Obtención del fitness dada una composición.
  - Obtención de la aplicación/arquitectura con la estructura de un grafo para facilitar la manera en la que se implementa la latencia.

- **UtilityApplication**: clase que hereda de Application. Incluye los métodos y estructuras necesarias para realizar el pre-cómputo. Algunas de las variables más importantes que contiene esta clase son:

  - Una estructura que almacenará el valor mínimo y máximo los atributos QoS.
  - La matriz de calidad de serivio, la matriz de probabilidad y la matriz de utilidad normalizada descritas en la sección 4.3.
  - La estructura que almacena la utilidades de los proveedores de cada servicio descrita en la sección 4.2.

    - Una estructura que almacena la latencia estimada de cada proveedor de cada servicio.
    - Una estructura que almacena las restricciones normalizadas utilizadas dentro de la llamada del genético.

  Los métodos más importantes son los descritos a continuación:

    - Calcular el máximo y mínimo de cada atributo QoS dentro del servicio.
    - Calcular la matriz de calidad de servicio.
    - Calcular la matriz de probabilidad.
    - Calcular la matriz de utilidad normalizada.
    - Obtener el valor agregado máximo y mínimo de la aplicación/arquitectura dada utilizando las fórmulas de agregación dadas por la sección 2.1.
    - Obtener la normalización de las restricciones.
    - Calcular la latencia estimada del proveedor por servicio.

- **UtilityGenotype**: es la clase que implementa las operaciones para conseguir el fitness del genotipo dado.
- **UtilityProblem**: en ella se establece la codificación del cromosoma.
- **Utility**: en ella se implementan todos los métodos resolutores para los métodos basados en la utilidad. Dispone de los dos resolutores: UR y UGA.
    - Resolución por UGA: Será necesario realizar los siguientes pasos para poder devolver una solución en forma de composición:
        1. Crear la UtilityApplication. En ella se realizan los pre-cómputos necesarios.
        2. Llamar a prepareEngine en el que se realiza la resolución de la aplicación de utilidad dada mediante el algoritmo genético. Esto devolverá el individuo de la población con mejor fitness y las estadísticas de

la ejecución del algoritmo genético (peor fitness, fitness medio, mejor fitness, tiempo de ejecución, número de generaciones, etc.).

3. Conversión del mejor individuo a composición.
4. (Opcional) Obtención del fitness de la composición dada.

Los pasos que se deben seguir para poder ejecutar los métodos basados en la utilidad son los que se listan a continuación:

- Generar la aplicación. Para ello, disponemos de diversos constructores cuyas aplicaciones serán creadas con una mayor o menor cantidad de parámetros atendiendo a las necesidades. Algunos de los parámetros que se pueden pasar son:
    - Un mapa con los patrones arquitectónicos aplicados y sus respectivas probabilidades en la generación de la aplicación.
    - La lista de atributos de calidad de servicio tenidos en cuenta.
    - El número de proveedores.
    - El número de servicios.
    - El rango de proveedores que puede tener cada servicio. No confundir con número de proveedores. Este no es más que la cantidad de proveedores distintos que se van a generar. El rango define el número de proveedores distintos que un servicio puede hospedar.
    - La semilla para generar la aplicación y hacer repetibles los experimentos.
- Generar la aplicación basada en la utilidad. Será necesario pasarle como mínimo una Aplicación en el constructor y el número de grados. En el caso de no pasar como parámetro las restricciones impuestas, se dará por supuesto que se busca la mejor composición.
- Llamar al resolutor en el cual se almacenarán los datos de la ejecución en una estructura para posteriormente exportarlos a un archivo CSV.

CAPÍTULO 6

Experimentos

En este apartado se mostrarán los resultados obtenidos de los experimentos desarrollados para comprobar el correcto y esperado funcionamiento del algoritmo. Para ello, se han comparado los resultados obtenidos con nuestro método y con los obtenidos con otros dos métodos que son aceptados dentro de la comunidad científica como buenas soluciones para la resolución de este problema de la composición de servicios. Los métodos aplicados en esta sección y sus abreviaciones, por simplicidad, serán los siguientes:

- Algoritmo genético completo (GA): Es el método descrito en [3]. Es el algoritmo genético clásico por excelencia. Cada uno de los índices de los genes se corresponden con el servicio i-ésimo. El genotipo será de longitud número de servicios y el gen elegido será el del índice del proveedor j-ésimo. El fitness será el de obtener los valores agregados de cada uno de los atributos de cada individuo y normalizándolo entre el máximo y mínimo multiplicado por su peso. Este método acepta restricciones.

- Algoritmo genético con utilidad (UGA): Se trata del método descrito en este trabajo, véase la sección 4.3. Este método acepta restricciones.

- Divide y vencerás (DyV): Es el método descrito por Pozas et al. en [6]. Un divide y vencerás que aplica un algoritmo genético en ciertos patrones para obtener la mejor solución. Este método no acepta restricciones.

- Nivel de utilidad por proveedor (UR): Se trata del método basado en la utilidad que da cada proveedor dentro del servicio. Se trata del método descrito en la sección 4.2.

6.1. Experimento 1

En este experimento compararemos las soluciones dadas por el DyV clásico y el UR. Para ello se comparará:

- Los tiempos de ejecución de cada uno de ellos para la misma aplicación.
- La calidad de las soluciones para la misma aplicación ejecutada con cada uno de ellos.

En la figura 6.1 se pueden ver los tiempos de ejecución para cada uno de estos algoritmos para una aplicación con 500 servicios y distinto número de proveedores (100, 200 y 300). Para el mismo número de proveedores se han realizado 10 ejecuciones, ya que el método descrito por Pozas et al. en [6] hace uso de un algoritmo genético. Podría darse casos en los que la solución fuera las de un mínimo local. Para paliar que la comparativa se vea perjudicada en caso de obtener una mala solución, se hace dicho número de ejecuciones repetidas. El fitness del método DyV será el de la media de las 10 ejecuciones. La figura 6.2 muestra los resultados para una aplicación con 1000 servicios y mismo número de proveedores que la ejecución con 500 servicios. Los tiempos de ejecución suelen diferir en un segundo entre ambos métodos. Para conocer más detalles de las cifras de los experimentos, véase la tabla 6.1.

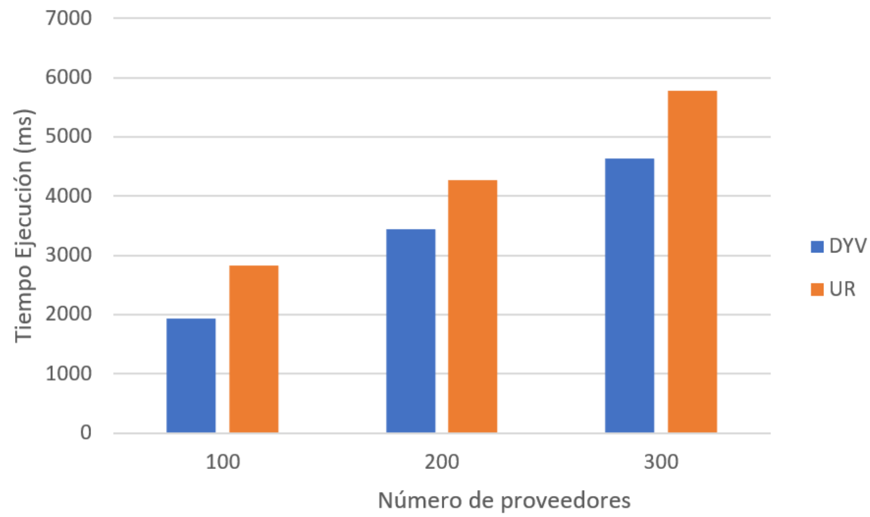


Figura 6.1: Evolución del tiempo de DyV y UR para 500 servicios y diferente número de proveedores.

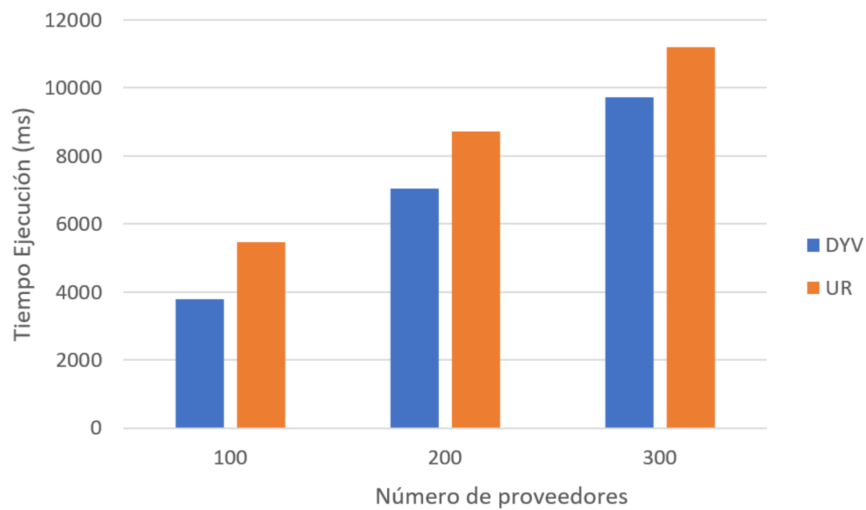


Figura 6.2: Evolución del tiempo de DyV y UR para 1000 servicios y diferente número de proveedores.

Las figuras 6.3 y 6.4 muestran la evolución del fitness para los experimentos de las figuras 6.1 y la 6.2 respectivamente. En ellas se puede observar que, aunque el tiempo de ejecución es ligeramente superior, la calidad de las soluciones dadas por UR es superior al DyV.

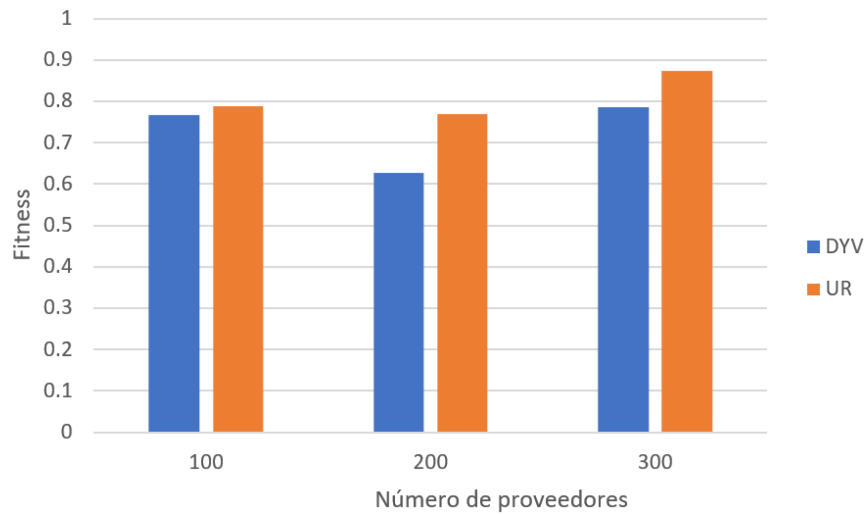


Figura 6.3: Evolución del fitness de DyV y UR para 500 servicios y para diferente número de proveedores.

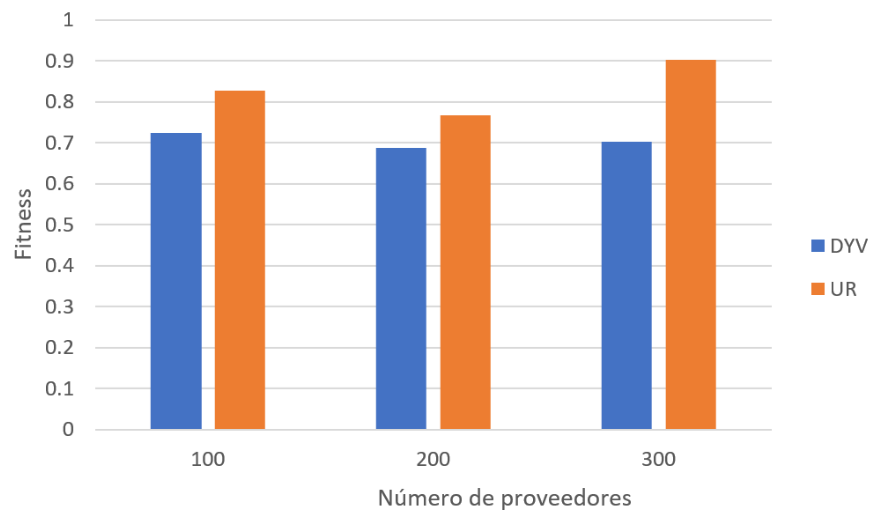


Figura 6.4: Evolución del fitness de DyV y UR para 1000 servicios y para diferente número de proveedores.

Servicios	Método	Proveedores	Media tiempo	Media fitness
500	**DyV**	100	1938.23548	0.76687777
		200	3437.39215	0.6272593
		300	4630.48989	0.78678313
	UR	100	2824.17255	0.78858998
		200	4275.49266	0.76895504
		300	5786.60175	0.87244004
1000	**DyV**	100	3794.90978	0.72506355
		200	7041.4011	0.68736531
		300	9718.48111	0.70210033
	UR	100	5474.82836	0.82795962
		200	8715.935	0.76783846
		300	11186.8042	0.90271424

Tabla 6.1: Tabla de los tiempos y fitness de los experimentos con DyV y UR

6.2. Experimento 2

En esta segunda ronda de experimentos se comparan los dos métodos que soportan restricciones. Para todos los experimentos, se ha realizado la resolución de la mejor composición sobre la misma arquitectura. A su vez, para la misma arquitectura se han realizado varias ejecuciones para obtener la media de sus fitness para evitar, en el caso de obtener como resultado un mínimo, que los resultados no se vean desfavorecidos.

Al comparar los tiempos de ejecuciones de GA y UGA observamos que el genético con utilidad (UGA) converge más rápido que el genético completo (GA). Se pueden encontrar diferencias de hasta 130 segundos, tal como se puede ver en las figuras 6.5 y 6.6.

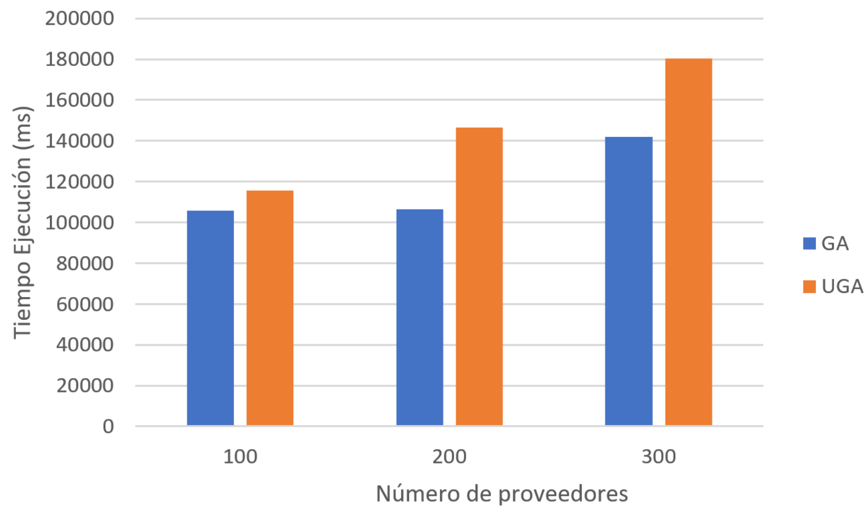


Figura 6.5: Evolución del tiempo de GA y UGA para 500 servicios y diferente número de proveedores.

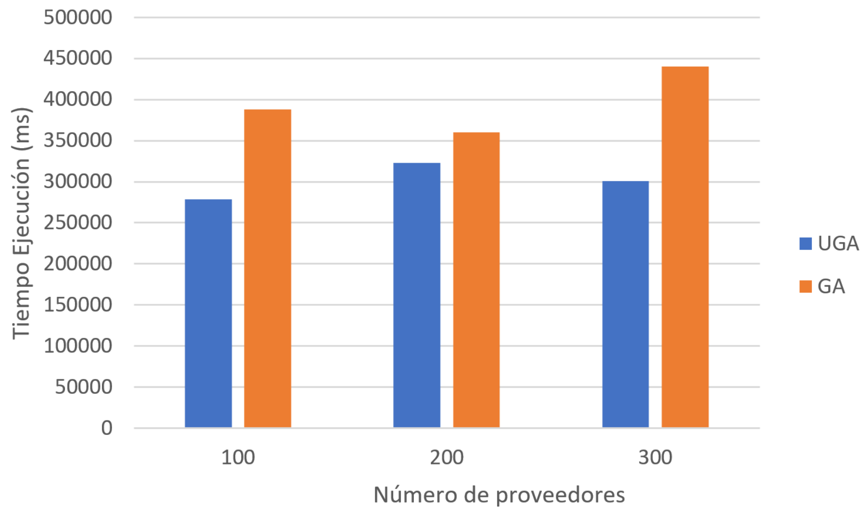


Figura 6.6: Evolución del tiempo de GA y UGA para 1000 servicios y diferente número de proveedores.

En las figuras, 6.7 y 6.8, podemos observar que a costa de un menor tiempo de ejecución para la resolución de la composición por el método UGA, se pierde cierta calidad de la solución, saliendo la del GA con mayor fitness.

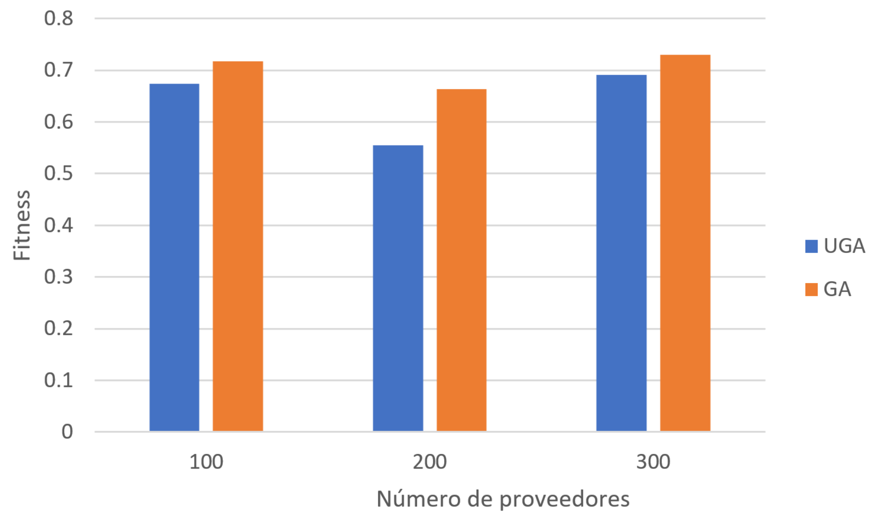


Figura 6.7: Fitness de GA y UGA para 500 servicios y diferente número de proveedores.

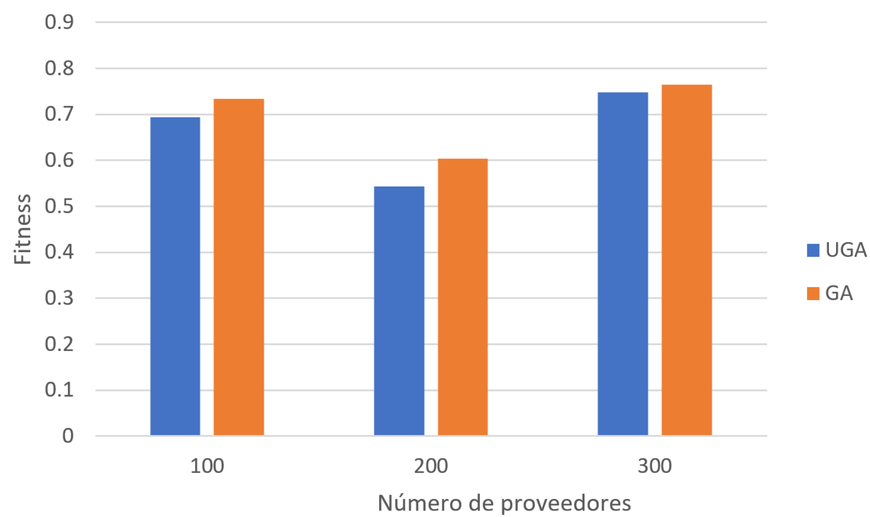


Figura 6.8: Fitness de GA y UGA para 1000 servicios y diferente número de proveedores.

Servicios	Método	Proveedores	Media tiempo	Media fitness
500	**GA**	100	115744.754	0.71766298
		200	146516.754	0.66329859
		300	180291.959	0.73049268
	UGA_15	100	105802.531	0.67434567
		200	106307.68	0.55493614
		300	141920.825	0.69092097
1000	**GA**	100	388313.02	0.73392344
		200	359966.919	0.6040857
		300	440517.649	0.76402469
	UGA_15	100	278267.838	0.69308999
		200	323004.62	0.54254646
		300	301036.507	0.7479385

Tabla 6.2: Tabla de los tiempos y fitness de los experimentos con GA y UGA

6.3. Discusión de los experimentos

Los experimentos 1 y 2 buscan en todo momento la mejor composición. Se puede observar que el tiempo crece exponencialmente si se aplica un algoritmo genético, mientras que el tiempo empleado por los métodos DyV y UR crecen linealmente. Además, otro resultado muy interesante es que en la búsqueda de la mejor composición los métodos lineales ofrecen para las mismas condiciones y aplicaciones un mejor fitness en muchos casos. Los algoritmos genéticos tienden a encontrar una solución lo suficientemente aceptable en un tiempo rápido, pero llegar a perfeccionarla le cuesta un tiempo excesivo y más aún cuando se trata de un genotipo de una longitud tan grande. Por tanto, el uso de los genéticos deberá estar simplemente limitado a cuando se utilicen restricciones.

Viendo los datos obtenidos para DyV y UR buscando la mejor composición, la diferencia entre un método u otro difiere aproximadamente en 1 segundo. Este tiempo

es despreciable cuando hablamos de aplicaciones de tal calibre, con un espacio de búsqueda de hasta 1000^{300} posibles configuraciones. La diferencia que se puede encontrar de hasta un 20 % cuando usamos un DyV y UR para 300 proveedores y 1000 servicios, será motivo más que suficiente para decantarse por el método UR.

Cuando tratamos con los dos métodos que aplican un algoritmo genético, se puede ver claramente que UGA es más rápido, en términos de tiempo de ejecución, que GA. Por otra parte, la solución dada por UGA es de menor calidad que la de GA. Pero a diferencia de cuando mencionamos en el párrafo anterior de decantarnos por la solución que tarda más debido a que la calidad de dicha solución es mejor, en este caso es distinto. Cuando tratamos UGA y GA, la diferencia puede llegar a verse incrementada hasta en 130 segundos, por esto, dependiendo de lo que priorice el usuario, deberá elegirse un método antes que otro:

- Si la prioridad es una mejor solución tratando restricciones sin importar el tiempo empleado: GA
- Si la prioridad es una solución aceptable tratando restricciones y el tiempo sí que es un factor decisivo: UGA

Finalmente, cuando nos referimos a la búsqueda de la mejor composición, no cabe duda de que la propuesta preferente es la del uso del UR. DyV da una solución de menor calidad a costa de tardar menos, pero dicho tiempo es despreciable. Por otro lado, los algoritmos genéticos no llegan a dar la misma calidad que estos dos métodos, por no hablar de sus desorbitados tiempos de ejecución.

CAPÍTULO 7

Conclusiones y líneas futuras

Acabadas todas las secciones que introducen el problema a resolver y los resultados que este genera, se realizará un pequeño análisis conclusivo con las recapitulaciones que podemos deducir de las mismas.

La resolución por utilidad es un método que trabaja en dos modos distintos:

- Aceptando restricciones. Será necesaria una primera fase donde segmentamos los datos, posteriormente buscamos la utilidad más cercana y si es posible, que supere la restricción. Finalmente lo convertimos en composición.

- No aceptando restricciones. En el que simplemente se asignará al servicio el proveedor con mayor utilidad sin tener en cuenta la arquitectura subyacente a esta.

Uno de los problemas que se plantean en todos los trabajos relacionados en el gran espacio de búsqueda que esto supone, y por consiguiente, el elevado tiempo de ejecución. Gracias a las técnicas aplicadas, podemos resolver dos tipos de problemas en un tiempo muy adecuado. Para aquellos que no presenten restricciones utilizaremos el método de utilidad por proveedor (UR). Si ya se requiere el uso de restricciones, será necesario utilizar uno de los dos algoritmos genéticos presentados, que dependiendo de las preferencias en cuanto a tiempo por parte del usuario, se usará UGA o GA.

También se ha podido observar que el heurístico aplicado para la latencia y la simplificación para el ancho de banda otorgan muy buenos resultados, a pesar de no tratarse de métodos que sí que empleen cálculos exactos como el caso de GA. En el algoritmo genético (GA), la latencia se calcula dentro del propio genético, ya que el propio genotipo de este método (GA) permite esto. Tal como se señaló, cada índice es un servicio en concreto y el número es el proveedor seleccionado. Aplicando la función de agregación para esa composición en particular, será posible tener un dato certero de la latencia y poder utilizarla para discriminar soluciones basadas en valores exactos. Por esta misma razón, y dadas las limitaciones que impone el método de utilidad, consideramos que los resultados obtenidos por el heurístico creado en este trabajo para tratar la latencia y la simplificación para el ancho de banda, dan unos resultados más que excelentes comparado con un método que tiene la ventaja de poder comprobar el valor exacto y no uno estimado.

El próximo reto y las líneas futuras que se podrían abordar son las listadas a continuación:

- Entontrar mecanismos para poder obtener una latencia real y no la aproximada. Para esto, un punto de partida podría ser el calcular en caliente dentro de la llamada del genético la latencia real, ya que en ella disponemos un cromosoma que puede ser traducido a composición. De esta forma, al saber cada par de proveedores que se elegiría, sería posible calcular la latencia real. Un posible problema que esto traería es que el tiempo de ejecución del genético podría dispararse.

- Otra línea que puede abrirse sería la de no realizar las simplificaciones que se hacen con el ancho de banda, es decir, la de suponer que será el usuario el que pondrá los mecanismos necesarios para poder establecer la conexión y que los datos no se pierdan. El principal motivo de descartar la idea temporalmente es que queríamos dar siempre una solución. Es muy probable que no pudiera existir conexión posible debido a que nuestras pruebas utilizan un enorme número de servicios y es cuestión de probabilidad que dos nodos conectados no tengan ninguna interfaz en común.

Bibliografía

[1] Batista, M., Moreno-Pérez, J., Moreno-Vega, J.: Algoritmos genéticos. una visión práctica. Números, ISSN 0212-3096, Nº. 71, 2009 (Ejemplar dedicado a: Darwin) (01 2009)

[2] Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for qos-aware web service composition. In: 2006 IEEE International Conference on Web Services (ICWS'06). pp. 72–82 (2006). https://doi.org/10.1109/ICWS.2006.69

[3] Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. GECCO '05, Association for Computing Machinery, New York, NY, USA (2005), https://doi.org/10.1145/1068009.1068189

[4] Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. Journal of Web Semantics pp. 281–308 (2004). https://doi.org/https://doi.org/10.1016/j.websem.2004.03.001

[5] Mardukhi, F., Nematbakhsh, N., Zamanifar, K., Barati, A.: Qos decomposition for service composition using genetic algorithm (2013)

[6] Pozas, N., Durán, F.: On the scalability of compositions of service-oriented applications. In: Hacid, H., Kao, O., Mecella, M., Moha, N., Paik, H.y. (eds.) Service-Oriented Computing. pp. 449–463. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-91431-8$_2$8

[7] Wilhelmstötter, F.: Jenetics, library use's manual 5.1 (11 2019), `https://jenetics.io/`

[8] Yu, T., Lin, K.J.: Service selection algorithms for composing complex services with multiple qos constraints. In: Benatallah, B., Casati, F., Traverso, P. (eds.) Service-Oriented Computing - ICSOC 2005. pp. 130–143. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11596141$_1$1

[9] Yuan, Y., Zhang, W., Zhang, X., Zhai, H.: Dynamic service selection based on adaptive global qos constraints decomposition. Symmetry p. 403 (2019)

[10] Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Transactions on Software Engineering pp. 311–327 (2004). https://doi.org/10.1109/TSE.2004.11

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga