Proyecto Fin de Carrera

Ingeniería Informática

Curso 2009/2010

Análisis teórico-práctico de métodos de inferencia filogenética basados en selección de modelos y métodos de superárboles (ANEXOS)

Autor:

Jorge Álvarez Jarreta

Bajo la dirección de:

Roberto Blanco Martínez
Elvira Mayordomo Cámara

Departamento de Informática e Ingeniería de Sistemas
Área de Lenguajes de Sistemas Informáticos
Centro Politécnico Superior
Universidad de Zaragoza

Septiembre de 2010

Índice general

Índice de figuras

Índice de tablas

A

Diagrama de Gantt

En la figura A.1 se puede ver el diagrama de Gantt correspondiente a este proyecto. Para aquellos menos familiarizados con esta herramienta, en dicho diagrama se refleja el tiempo dedicado a cada una de las tareas de un proyecto.

Poniendo un poco en contexto al lector, no debe sorprender la longitud en meses que ha durado este proyecto, pues durante el primer cuatrimestre cursé 7 asignaturas, la mayoría de 6 créditos, y durante el segundo, 3. Además, he trabajado con una beca de colaboración a lo largo del año, y he participado en un proyecto pequeño con vistas a unas jornadas nacionales a finales del segundo cuatrimetre. A continuación se va a comentar un poco las distintas tareas y el por qué de su orden y duración.

Yendo de arriba a abajo, en primer lugar se pueden ver la reuniones que han tenido lugar compuestas por los miembros del grupo de bioinformática. Se realizaban semanalmente durante aproximadamente 1 hora. En principio el día de la semana establecido eran los lunes, por lo que se ha mantenido así reflejado en el diagrama aunque, como es obvio, alguna tuvo que ser pospuesta por motivos diversos. En estas reuniones cada miembro exponía el trabajo que había realizado durante esa semana (o varias, según el caso) y ofrecía sus conocimientos a los demás miembros del grupo, así como acudía a ellos en busca de ayuda con dudas o problemas que le hubiesen surgido.

Como ya se ha comentado varias veces a lo largo de la memoria, la fase de documentación ha sido larga y continuada prácticamente a lo largo de todo el proyecto, de ahí su extensión.

Aunque la parte central del proyecto se basa en el sistema de inferencia filogenética, el proyecto se inició, tras la debida adquisición de conocimientos básicos, con el desarrollo del método de construcción de superárboles de corte mínimo. Debido a la carga docente del cuatrimestre, no me fue posible dedicarle todo el tiempo que me habría gustado diariamente, de ahí que las fases se alarguen tanto en el tiempo.

En los últimos meses, y ya con menos trabajo que atender, se ha realizado un trabajo más intensivo, sobre todo en el mes de Julio, para lograr completar el sistema de inferencia filogenética. Hay que indicar, aunque no se ha reflejado en el diagrama, que se ha realizado de forma paralela un

artículo sobre este tema con vistas a llevarlo a un congreso internacional. Actualmente dicho artículo esta a punto de ser completado.

Finalmente, y dado que la fase de pruebas del sistema requería de varias horas de ejecución por cada prueba, se empezó a redactar esta memoria.

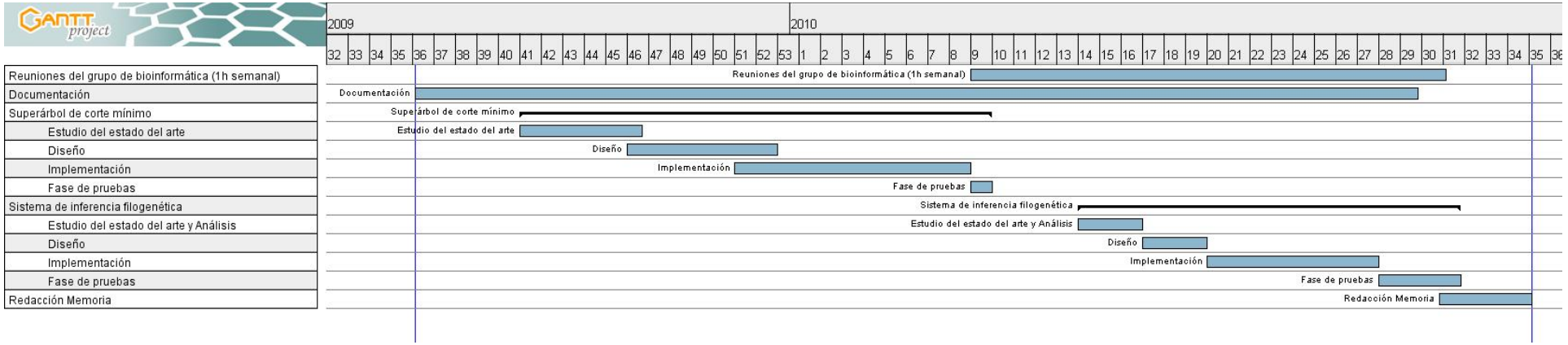

Figura A.1: Diagrama de Gantt.

B
Fundamentos biológicos

A lo largo de este apéndice se van a explicar todos aquellos términos y conceptos que se han considerado necesarios para la completa comprensión del trabajo realizado en este proyecto. Antes de continuar me gustaría agradecer a uno de mis directores, Roberto Blanco, su consentimiento para usar la memoria de su proyecto de fin de carrera [6] como inspiración y base para la realización de este capítulo.

B.1 Base biológica

Se ha considerado oportuno hacer una breve pausa en la mente informática del lector para mostrar el proyecto desde el punto de vista de un biólogo, permitiendo así apreciarlo en su conjunto y comprender mejor el contexto del trabajo.

El ADN mitocondrial humano, elemento tratado a lo largo de este proyecto, (ADNmt de ahora en adelante) es un elemento biológico muy interesante desde muchos puntos de vista. Al estar dentro de las mitocondrias y ser independiente del ADN celular es centro de muchas teorías y controversias respecto al origen o inclusión de estos corpúsculos en la célula (y gracias a los cuales hoy estamos nosotros aquí). Pero ese no es el tema que atañe a este proyecto. El ADNmt posee una tasa de cambio muy elevada, lo que lo hace idóneo para el estudio de individuos y su estrecha relación, como miembros de una misma especie. Además, aunque queda algo lejos de los objetivos planteados, el ADNmt está muy ligado a ciertas enfermedades que provocan una muerte prematura en los individuos que las padecen.

El ADNmt ha sido largamente estudiado, y actualmente se sabe que lo forman entre 16557 y 16576 nucleótidos, conteniendo 37 genes distintos, de los cuales 13 tienen como objetivo la creación de proteínas, 22 codifican ARNt (ARN transferente o de transferencia) y, cómo no, las dos unidades que conforman el ribosoma (ARNr). Hay que indicar que el ADNmt es circular, a diferencia del ADN celular, del que todo el mundo conoce su estructura de doble hélice. Por tanto, el ADNmt posee una región de control, también conocida como bucle D, que no tiene como objetivo la codificación sino la unión para conformar dicha estructura. Esta zona contiene dos regiones hi-

pervariables: HVR_1 y HVR_2, las cuales pueden determinar el haplogrupo de un organismo [34].

Un haplogrupo es una agrupación de haplotipos, los cuales representan conjuntos de polimorfismos con alguna característica estadística común. La clasificación y determinación de haplogrupos tiene como fundamento el detectar estas características comunes, las cuales pertenecerán de forma única a la familia en cuestión. Los haplogrupos son muy importantes en el estudio de la relación filogenética entre individuos, pudiendo determinar el punto de origen del linaje que se esté estudiando. En el caso del ser humano, el ancestro matrilineal común más reciente del ser humano se ha denominado "Eva mitocondrial" y vivió hace unos 140.000 años. El ancestro común más reciente en global, incluyendo la herencia patrilineal, es mucho más reciente.

Para terminar este apartado, comentar que la gran variabilidad del ADNmt se puede considerar fundamental para poder realizar una clasificación en haplogrupos de forma fiable. Para dar cuenta de dicha fiabilidad, por ejemplo, se estima que entre dos seres humanos cualesquiera existen únicamente entre 50 y 70 nucleótidos diferentes, es decir, un 0'42 % del total de nucleótidos del ADNmt, lo que es una proporción muy elevada.

B.2 Introducción a la bioinformática

La bioinformática se puede considerar una rama de reciente aparición en la informática cuyo objetivo es el uso de la informática para la resolución de problemas biológicos. Su necesidad se ha visto acentuada en los últimos años debido a la magnitud y complejidad que están adquiriendo ciertas investigaciones referentes a la biología, que hacen intratable su resolución de forma manual [23, 15].

En concreto en este proyecto se han tratado temas del área de análisis de secuencias, más en concreto, sobre construcción de filogenias. Para poder desarrollar el estudio de filogenias es indispensable disponer de secuencias previamente alineadas. El alineamiento de secuencias no ha sido objetivo de este proyecto, por lo que se ha trabajado con las técnicas que se estudiaron y desarrollaron en anteriores proyectos de fin de carrera [6]. Para dar una idea al lector de en qué consisten estas técnicas, el alineamiento de secuencias es una operación que establece equivalencias entre los distintos caracteres de las secuencias introducidas. Haciendo la suposición de que entre dichas secuencias existe un parentesco, pretende detectar aquellos hechos que las separan.

B.2.1. Filogenética

La filogenética se puede considerar una ciencia, que tiene como objetivo la clasificación de las especies en base no a características fenotípicas si no a su relación evolutiva. Si el lector esta familiarizado con estos temas puede

que vea en esta definición muchas coincidencias con la cladística, y no se equivoca: a menudo filogenética y cladística son tratadas como sinónimos por tener el mismo objetivo.

La cladística tiende a definir distintos tipos de grupos de especies. Los más conocidos son:

- Grupo monofilético o clado: compuesto por un organismo y todos sus descendientes. Al igual que pasaba con "Eva mitocondrial", la raíz u origen de este grupo es el ancestro común más reciente del mismo.
- Grupo parafilético: compuesto por aquellos organismos cuya raíz u origen no incluye a todos los descendientes.
- Grupo polifilético: compuesto por varios grupos monofiléticos no solapados. Dada su complejidad suele desaconsejarse su uso.

Como se habrá podido fijar, en la definición de los grupos se ha mencionado la palabra raíz en varias ocasiones. Eso es debido a que estos grupos normalmente se representan en árboles biológicos donde se expresa la relación de parentesco entre organismos o especies. Estos árboles se conocen como árboles filogenéticos. Aunque normalmente, sobre todo en la cladística, estos árboles suelen ser binarios, la aridad o número de subárboles que se derivan de cada nodo interno puede ser mayor que 2.

B.2.2. Modelos evolutivos

Los modelos evolutivos, también conocidos como modelos de substitución, son modelos matemáticos que se crearon para definir la probabilidad de cambio entre los distintos nucleótidos en secuencias de ADN o ARN, o aminoácidos, en el caso de proteínas. Su objetivo es intentar explicar la evolución que han sufrido dichas secuencias a lo largo del tiempo. Incluso existe la opción de que un cambio que se haya producido se deshaga con el paso del tiempo; es decir, se contempla la posibilidad de reversión de un cambio anterior.

Estos modelos se rigen por una serie de parámetros, quedando definidos por los valores que se les parcial o totalmente, de forma que los que queden libres se ajusten a las secuencias de entrada cuando se evalúe el modelo en cuestión. Por ejemplo, algunos modelos prefijan que el cambio entre nucleótidos en secuencias de ADN o ARN sea equiprobable, mientras que otros postulan que las transiciones (cambio entre nucleótidos del mismo tipo) son más frecuentes que las transversiones (cambio entre nucleótidos de distinto tipo). Recordar que con tipos se hace referencia a la división entre purinas, que son los nucleótidos A y G, y las pirimidinas, C y T. Se ha de tener en cuenta que todos los modelos necesitan que se les indique la distribución inicial de las frecuencias de los distintos nucleótidos o aminoácidos.

B.2.3. Selección de modelos

Existe una extensa variedad de modelos actualmente, y su elección, lejos de ser insignificante, puede acarrear serios problemas en trabajos de análisis o estudio de secuencias o proteínas. Principalmente, una mala elección en un modelo evolutivo puede involucrar relaciones incorrectas entre especímenes o, en el mejor de los casos, un árbol de peor calidad [22, 24, 26, 28, 29, 33, 40, 43, 49].

Hay que tener siempre en cuenta que todos los modelos evolutivos tratan de explicar la relación real entre los individuos, pero dada la complejidad y desconocimiento que se posee actualmente de la evolución cabe esperar que ni el mejor de los árboles sea fiel a la realidad [39]. Por tanto, nunca se debe olvidar que la filogenética pretende obtener filogenias próximas a la realidad, pero en ningún momento no se puede estar seguro de que la coincidencia sea total.

Aunque la importancia de su elección levanta opiniones diversas entre los investigadores, debido a la imprecisión ya comentada, se intenta demostrar la efectividad de estos modelos comparando los resultados que se obtienen con conjuntos de pruebas creados de forma artificial, de los cuales se conoce su relación evolutiva "real" y, por tanto, su modelo evolutivo correspondiente.

Existen varios métodos aplicables a un árbol filogenético para conseguir un valor que pueda indicar qué modelos se aproximan más a la realidad intrínseca de los datos. Un método muy aplicado es el de máxima verosimilitud. Este método tiene como objetivo deducir los datos de entrada a partir de los resultados observables [39, 22, 42]. Asignando una probabilidad a las distintas mutaciones que ha podido sufrir una secuencia, estima la distribución de probabilidad del espacio de árboles. Es uno de los métodos más flexibles pero, a su vez, resulta uno de los más costosos, computacionalmente hablando. Se sospecha que puede tratarse de un problema NP-completo; pero así como para otros método ha podido demostrarse, para el método de máxima verosimilitud sigue siendo únicamente una suposición.

Hay que tener en cuenta que, pese a ser una aproximación más sencilla y potente que otros métodos, tiene el inconveniente de valorar mejor a los modelos más complejos (con un mayor número de parámetros libres). Para compensar esta deficiencia se han desarrollado los criterios de información, cuya base se sustenta en penalizar aquellos modelos de mayor complejidad. Así, mediante una combinación de ambos parámetros (complejidad y verosimilitud), se puede escoger un modelo evolutivo equitativo, dejando ya a decisión del investigador el sesgo o no del criterio que ambos métodos establecen. Como parte de los criterios de información se pueden encontrar el de Akaike (conocido como AIC) y el bayesiano (conocido como BIC).

C

Evaluación de jModelTest

C.1 Descripción de la aplicación

jModelTest [25] es una herramienta que permite realizar la evaluación de modelos evolutivos a un alineamiento de entrada. Esa aplicación se ha realizado sobre Java, y ofrece un entorno con un manejo muy sencillo que, entre otras cosas, transforma las peticiones del usuario en comandos que ejecuta con Phyml para la evaluación de modelos evolutivos.

Ofrece varias opciones a la hora de evaluar los modelos una vez proporcionado un alineamiento, con varios conjuntos de modelos evolutivos, donde el mayor está formado, en la versión 0.1, por 88 modelos de distinta complejidad. Además de realizar la evaluación mediante el método de máxima verosimiltud, ofrece otras opciones interesantes como la aplicación del criterio de información de Akaike o el bayesiano.

Actualmente es una de las herramientas más usadas en el mundo de la bioinformática, y sigue en desarrollo. Pero, como se mostrará en la sección siguiente, en su diseño se han cometido algunos fallos que la convierten en una realmente ineficiente cuando el número de secuencias es relativamente grande, pudiendo llegar a ser intratable con alineamientos realmente grandes.

C.2 Estudio realizado

Dado que es la herramienta de referencia en numerosos artículos sobre evaluación de modelos, se consideró más que oportuno el estudiar su funcionamiento y limitaciones antes de lanzarnos a crear el sistema de inferencia filogenética ya expuesto.

En un primer momento, cuando se realizó una primera prueba con un alineamiento muy pequeño que venía de ejemplo con la aplicación, se detectó un primer inconveniente: los modelos son evaluados de forma secuencial. Con esto quedaba claro que, de no realizarse algún tipo de optimización que no se pudiese observar inicialmente, existía un componente que animaba a pensar en un fallo de rendimiento en las pruebas que se iban a realizar.

Dichas pruebas estaban formadas por alineamientos con distintos número de secuencias y distinto número de nucleótidos, siempre ejecutándose en el mismo ordenador y, aproximadamente, con la misma cantidad de recursos disponibles. En concreto, se seleccionaron 8 alineamientos distintos, 4 de 100 secuencias y 4 de 200 secuencias, con 4000, 8000, 12000 y 16000 nucleótidos. El objetivo que se pretendía alcanzar con estos ficheros era detectar su efecto en el coste temporal al incrementar el número de nucleótidos y el número de secuencias, tanto de forma independiente como conjunta.

Además, aprovechando los resultados que se fueron obteniendo, se recopilaron datos para mostrar cómo afecta a la selección del modelo final el criterio aplicado, mostrando el modelo seleccionado mediante máxima verosimilitud, el criterio de información Akaike (AIC) y el criterio de información bayesiano (BIC).

Los resultados se han recopilado en la tabla C.1 y las imágenes C.1 y C.2.

Nº Secs.	Nº Nucl.	Tiempo	Modelo Máx. Ver.	Modelo AIC	Modelo BIC
100	4000	01h 27m 33s	GTR+I+G	TIM2+I	TrN+I
100	8000	01h 42m 32s	GTR+G	TIM2	TrN
100	12000	03h 10m 26s	GTR+I	TIM1	TIM1
100	16000	04h 23m 28s	GTR+I	TIM3+I	TIM3+I
200	4000	05h 15m 37s	GTR+I	TIM2+I	TrN
200	8000	08h 11m 58s	GTR+I	TIM3+I	TrN
200	12000	11h 57m 13s	GTR+I	TIM3+I	TIM3
200	16000	17h 20m 34s	GTR+I	GTR+I	TIM3+I

Tabla C.1: Alineamientos con el coste temporal y los modelos seleccionados por los distintos criterios.

En la tabla C.1 se puede ver cómo el método de máxima verosimilitud tiende a seleccionar los modelos más complejos y generales (GTR o una de sus versiones ampliadas) mientras que los criterios de información, aparte de no coincidir en muchas ocasiones (de ahí su interés), tienden a escoger modelos algo menos complejos, donde el número de parámetros libres afectan mucho menos a su ratio final.

Tras observar los resultados, se puede concluir que esta aplicación solo es recomendable usarla para alineamientos pequeños, pues el incremento, sobre todo en el número de secuencias, tiene como consecuencia el crecimiento, casi exponencial, de su coste temporal, por lo que no es descabellado pensar que se podrían tener actualmente alineamientos que esta herramienta tardase varios años en tratar. De ahí la importancia de desarrollar un sistema como el expuesto a lo largo de esta memoria.

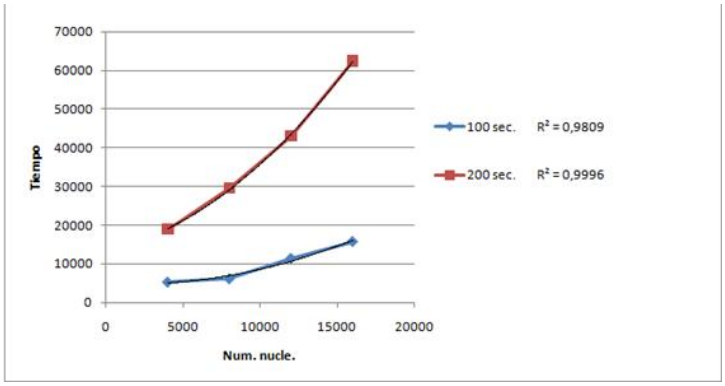

Figura C.1: Gráfica de estudio de costes temporales con división por secuencias.

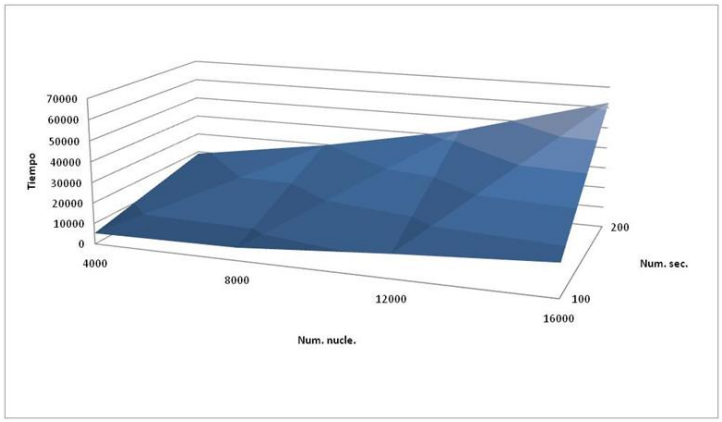

Figura C.2: Gráfica de estudio de la evolución del coste temporal de jModelTest según se incremente el número de secuencias o de nucleótidos.

D

Manual de usuario: *Sistema de inferencia filogenética*

Se han desarrollado scripts que facilitan el manejo del sistema. Así, este manual, más que describir la forma de crear y ejecutar el sistema, se centra en el formato que han de tener los ficheros de entrada y el formato que tendrán los ficheros de resultados.

D.1 Ficheros de entrada

Para el correcto funcionamiento del sistema hay que incluir 3 ficheros de entrada en su versión actual, dado que el método de construcción de superárboles integrado en la última versión del sistema requiere de un superárbol base, como se comentaba en el capítulo Superárboles.

El primer fichero contendrá el alineamiento de las secuencias que se quieran estudiar en formato Phylip [2]. Este formato es ampliamente usado y conocido en bioinformática. Para 3 secuencias de 20 nucleótidos el contenido del fichero en este formato sería el siguiente:

```
  3  20
seq00001   AGTCATCTAT ATCTACGGTA
seq00002   -GATCACAGG TCTATCACCC
seq00003   TCTATCACCC TATTAACCAC
```

El segundo archivo necesario contiene los haplogrupos (o grupos de secuencias) para la división del alineamiento inicial. Se ha creado un formato propio para este fichero, indicando en una línea el identificador del grupo o haplogrupo precedido por la cadena "> " y en la siguiente línea, y separados por un espacio, los identificadores de cada una de las cadenas que pertenecen a dicho grupo. Deben estar todos en una línea, es decir, para que el formato sea correcto y no genere errores, los identificadores de las secuencias no pueden estar escritos en varias líneas. Para 2 grupos de 1 y 2 secuencias respectivamente, el fichero quedaría como sigue:

```
> A
```

seq00001
> B
seq00002 seq00003

El tercer fichero corresponde al superárbol base para la construcción del superárbol final. Se debe realizar en formato Newick [3], con o sin longitudes de las ramas, donde las hojas sean los identificadores de los grupos del fichero anterior. Siguiendo los ejemplos anteriores, una posible versión de este fichero sería:

(A, B);

D.2 Cómo montar y ejecutar el sistema

Ejecutando el script load.py se generarán todos los ficheros del sistema. Dicho script requiere que se introduzca la cadena "-d" como primer argumento si se desea activar la opción de debugging, es decir, que no se borren los ficheros temporales que vayan apareciendo durante la ejecución del sistema. Si no se desea incluir esta opción, simplemente pase al siguiente argumento. A continuación será necesario proporcionar el nombre del fichero con el alineamiento, después el fichero con los grupos de secuencias, y a continuación el fichero del superárbol base. Tras estos tres ficheros, se irán incluyendo una a una las posiciones para realizar los cortes en columnas o genes, separadas por un espacio. El último argumento será el número de bootstraps a generar por el sistema durante su ejecución.

Una vez generados todos los ficheros del sistema, solo es necesario ejecutar el script start.sh para comenzar la ejecución del sistema.

D.3 Ficheros de salida

El fichero principal, y más importante de los que se generan como resultado de la ejecución del sistema, tiene el mismo nombre que el superárbol base, solo que con el sufijo "_outtree.txt". En este fichero se encuentra el superárbol en formato Newick obtenido tras el estudio filogenético realizado.

A continuación se van a dar algunas pautas para comprender el contenido de los ficheros que se generan de forma temporal, para aquellos usuarios que ejecuten el sistema en modo debugging.

Prefijos: los ficheros con código alfanumérico poseen una o varias letras que lo preceden, indicando así su tipo o contenido.

- Las letras "hg" indican que el archivo pertenece al grupo cuyo identificador se encuentra tras estas dos letras.
- La letra "c" indica que el archivo pertenece al gen o grupo de columnas del identificador que la acompaña.

- La letra “l” indica que el archivo es un link soft.
- La letra “b” indica que el archivo pertenece a la fase de bootstrap.

Extensiones: únicamente indicar que los ficheros con extensión “phy” contienen alineamientos en formato Phylip. Los árboles generados, todos en formato Newick, se encuentran en ficheros con extensión “txt” y en su nombre se puede leer la palabra “tree”.

E

Selección de lenguaje de programación para el algoritmo de superárboles de corte mínimo

(INFORME INTERNO)

EL ANÁLISIS DE DIJKSTRA

Análisis de dos lenguajes de programación el algoritmo de Dijkstra, para poder elegir el más idóneo ante los nuevos retos biológicos

Jorge Álvarez Jarreta

Ciencia, Tecnología y Sociedad, Quinto curso, Centro Politécnico Superior, Universidad de Zaragoza. 534608@unizar.es

Abstract

Los lenguajes de programación son una de las herramientas esenciales de todo informático, puesto que suponen la base de todo programa. Aunque el análisis y diseño de cualquier problema se construyan bajo unos principios más matemáticos que informáticos, al final, a la hora de dar forma a esa solución, todo el trabajo hasta ese momento obtenido se intenta adaptar al lenguaje escogido, haciendo uso, en la medida de lo posible, de todas las ventajas que éste ofrece. Dado que es esencial conocer el lenguaje que va dar un mejor resultado ante el problema planteado, el análisis comparativo de lenguajes es, sin duda, una tarea necesaria y fundamental. La comparación entre dos o más lenguajes en su totalidad resultaría inabarcable, por lo que es necesario diseñar pruebas de aspecto general, pero que a su vez estén centradas en algún objetivo concreto que se desee analizar. Un caso de estudio interesante es la realización de medidas de rendimiento de dos lenguajes para problemas que requieran de estructuras de tipo grafo. Un problema muy conocido perteneciente a este grupo consiste en el planteamiento del cálculo del camino de coste mínimo de un vértice a todos los demás de un grafo. Edsger W. Dijkstra, un importante informático teórico, diseñó en 1959 un algoritmo para dar solución a este problema con un coste computacional excelente. Analizar el rendimiento de dos lenguajes para dicho algoritmo permitirá aportar suficiente información como para, ante problemas que requieran el uso de grafos, poder escoger entre estos dos lenguajes, y así poder obtener la mejor solución.

Palabras clave: Lenguaje de programación, grafos, algoritmo de Dijkstra, análisis, bioinformática.

1. Introducción

De unos años a esta parte, la tecnología está avanzando cada día más y más deprisa, lo que ha beneficiado enormemente a todas las ramas de la ciencia. Gracias a estos avances, se pueden analizar problemas más complejos, recopilar más información o llegar a sitios todavía no explorados hasta el momento. La biología es una de las ramas que se ha visto beneficiada por este avance, como por ejemplo, en la obtención de cada vez mayores cantidades de información. Existen hoy en día bases de datos con miles e incluso millones de secuencias de ADN de distintas especies. Uno puede hacerse a la idea de la cantidad en terabytes de información que esto supone, sobre todo teniendo en cuenta que se estima que el ADN humano contiene 3000 millones de pares de bases.

El avance tecnológico también ha permitido crear nuevos ordenadores capaces de solucionar problemas antes impensables. Un ejemplo claro de este avance queda reflejado en la ley que enunció Gordon E. Moore en 1965, en la cual decía que el número de transistores por chip se duplicaba cada año y que eso iba a seguir siendo así por, al menos,

10 años. [Moo65] A lo largo del tiempo se han ido realizando nuevas estimaciones que apuntaban a que dicha duplicación seguía produciéndose, aunque a ritmos diferentes. Actualmente esta duplicación se alcanza, aproximadamente, cada 20 meses. [Web1] Esto conllevó, entre otras cosas, a permitir resolver con los ordenadores problemas biológicos que antes resultaban impensables debido al coste temporal que suponían. Esta limitación implicaba la necesidad de que algunos informáticos se especializasen en este campo para poder responder adecuadamente a todas las necesidades que iban surgiendo por parte de los biólogos. De ahí nació lo que hoy se conoce como bioinformática.

Aún con todo, sigue faltando mucho por estudiar y descubrir. Por ello es necesario que en todo momento se usen las herramientas adecuadas para cada problema que surja, con el objetivo de optimizar lo máximo posible los resultados obtenidos. En la informática, una de las herramientas más importantes es el lenguaje de programación. Aunque éste intervenga en una de las últimas etapas del ciclo de vida del software, supone la base que permitirá que esa solución obtenida tras analizar el problema planteado, pueda ejecutarse en un ordenador. Es necesario, por tanto, poner especial atención y cuidado a la hora de su elección ante cada nuevo problema que surja. Desde la aparición de los primeros lenguajes de programación, siempre se ha intentado crear un lenguaje de programación que fuese el mejor para una serie de objetivos que los propios desarrolladores se marcaban. Hoy en día existen tantos lenguajes distintos, que el objetivo ha dejado de ser el crear un lenguaje óptimo para la solución diseñada, si no que es preferible (y bastante menos costoso) el realizar un análisis comparativo entre dos o más lenguajes existentes que, a priori, se considere que se adaptan mejor a las necesidades del problema. El objetivo de este análisis es evaluar, bajo una serie de criterios establecidos por el analista, qué lenguaje destaca en qué criterios y, con toda esa información, poder realizar la elección del lenguaje de programación óptimo.

2. Fundamentos técnicos

Es necesario saber que todo ordenador posee un repertorio de instrucciones básicas, muy sencillas, las cuales se traducen directamente a bits que el ordenador interpreta para llevar a cabo las tareas. El problema es que para problemas crecientes, este repertorio de instrucciones, por muy simple que resulte, se vuelve inmanejable, por lo que es necesario tener lenguajes de programación que nos abstraigan más del hardware que tenemos debajo a la hora de programar. Conforme los problemas son más y más complejos, son necesarios lenguajes que nos abstraigan cada vez más de la máquina, para facilitar así la tarea de diseño e implementación de aplicaciones que resuelvan dichos problemas. A esta abstracción se la denomina como nivel. Así, básicamente los lenguajes de programación se dividen en bajo nivel, como C, y lenguajes de alto nivel, donde en este grupo encontraríamos a Python y Ada.

Otro concepto con el que es necesario familiarizarse antes de continuar es el tipo de dato conocido como grafo. Un grafo es un mapa donde se representa la conexión entre distintos puntos denominados vértices o nodos. Los enlaces son reconocidos como aristas, que representan relaciones binarias. Normalmente estas aristas suelen llevar un peso asociado que representa el coste de pasar por ella para ir de un vértice a otro. Hay muchos problemas matemáticos, de distintos niveles de complejidad, planteados con grafos. Otra estructura también muy importante, sobre todo en problemas biológicos, son los árboles. Por definición, un árbol es un caso particular de un grafo, en el cual no hay ciclos (caminos que permiten desde un nodo, recorriendo una serie de aristas sin repetir ninguna, volver a sí mismo).

Por último, otro concepto necesario para comprender este documento en su totalidad es el paradigma. Un paradigma es un conjunto de reglas que "rigen" una determinada disciplina.[Web2] En nuestro caso, los paradigmas en programación son los patrones que siguen

los códigos que se desarrollan para cada lenguaje de programación. Existen el paradigma imperativo, funcional, lógico, orientado a objeto y muchos otros menos conocidos.

3. Los dos lenguajes candidatos

A la hora de elegir un lenguaje de programación puede ayudar el estudiar previamente el problema al que hemos de hallar solución. Con un primer análisis rápido de dicho problema nos debería bastar para poder escoger un paradigma adecuado. Una vez elegido el paradigma, la cantidad de lenguajes que podemos encontrar puede ser tan grande como paciencia tengamos en buscar. Normalmente siempre se acaba escogiendo un subgrupo, compuesto por los lenguajes más usados en los últimos años o aquellos novedosos que han surgido para deleite de aquellos a los que les guste conocer y analizar lenguajes de programación. En este caso, nos hemos decantado por el paradigma imperativo ya que, en cuanto a eficiencia se trata, suele contener los lenguajes que mejores resultados obtienen (aunque no se ha de tomar como regla general). De entre la enorme lista de candidatos, se han escogido Python y Ada. Un primer motivo es que esta elección permite no solo realizar el análisis de dos lenguajes de programación, sino que además revelará datos interesantes sobre la diferencia en cuanto a la eficiencia de un lenguaje interpretado y otro compilado. Otro de los motivos importantes es el control de excepciones, que ambos poseen. Esto no implica que vaya a ser usado en la implementación de la solución, pero permiten identificar de forma fácil y rápida los errores que puedan cometerse durante dicho proceso, lo que facilitará la tarea. Un último motivo es que Python es un lenguaje de reciente aparición, y que Google está anunciando y explotando, y Ada es el primer lenguaje de programación que enseñan en la carrera de ingeniería informática superior de la Universidad de Zaragoza.

3.1. Lenguaje interpretado: Python

Python es un lenguaje de propósito general, de alto nivel y multiparadigma. Por tanto, no solo admite el paradigma imperativo, sino que también permite trabajar con el paradigma orienta a objetos, el funcional, y unos cuantos más, menos relevantes. Sus diseñadores optaron por un hito principal a alcanzar con este lenguaje: la legibilidad. Con esto pretendían conseguir un lenguaje cuyo código, ante todo, fuese fácil de leer. Como ya se ha mencionado, Python es principalmente un lenguaje interpretado, aunque existe la opción de compilar los ficheros fuente en caso de que fuese necesario. Posee una librería estándar muy extensa y potente, con tipos de datos predefinidos como listas o diccionarios. Cabe destacar que posee control de excepciones, lo que hace que los códigos creados con este lenguaje puedan ser mucho más robustos y tener un comportamiento controlado ante posibles errores. Su uso más común es para la creación de scripts en aplicaciones web, aunque es aplicable a una gran cantidad de entornos. Python posee intérpretes para todos los sistemas operativos comunes, es decir, Windows, Mac OS y Linux. [Web2]

La forma con la que se suele relacionar la sencillez y potencia de un lenguaje es mediante la implementación de un programa que simplemente escriba por pantalla *"Hello world!"*. Esta aplicación, además, suele ser el primer ejercicio que se plantea resolver cuando se empieza a estudiar un lenguaje de programación, dada su simplicidad. En el caso de Python, el código necesario para crear la aplicación *Hello world* sería:

```
print "Hello, World!"
```

3.2. Lenguaje compilado: Ada

Ada es un lenguaje de programación de alto nivel, fuertemente tipado, que soporta tanto el paradigma imperativo con el orientado a objetos. El hecho de ser fuertemente tipado hace que se deba prestar especial atención a la hora de trabajar con varios tipos de datos, dado

que todo paso de información entre variables de distinto tipo deberá quedar indicado de forma explícita mediante una conversión al tipo destino, ya que de forma contraria el compilador generará un error y el código no será compilado. Una de sus características más importantes es que permite la creación de paquetes genéricos, muy útiles para reutilizar en otro tipo de problemas, ya que se pueden adaptar a las necesidades del mismo (en la medida que la definición del paquete lo permita). Otro de los puntos fuertes de este lenguaje es su sistema de control de excepciones, que permite realizar programas muy robustos. Además, posee unos paquetes muy eficientes que le hace ser un candidato idóneo para implementar sistemas de tiempo real. Originalmente este lenguaje fue creado a petición del departamento de defensa de los Estados Unidos. Su nombre fue dado en honor a Ada Lovelace, a la que se le ha atribuido en varias ocasiones el ser la primera programadora informática. Ada es un lenguaje muy usado, sobre todo, en sistemas de aeronáutica. [Web3]

Al igual que se ha dicho con Python, quizá una de las forma más claras de demostrar de forma sencilla lo bueno que puede llegar a ser un lenguaje es el crear el programa *Hello World*. Para el caso de Ada, el código sería el siguiente:

```
with Text_Io;
use Text_Io;

procedure Hello_World is
begin
   put ("Hello world!");
end Hello_World;
```

4. Ante qué se enfrentan

Muchos problemas biológicos tienen que ver con la conectividad que puede existir entre ciertos datos, lo que, en términos informáticos, implica la necesidad del uso de los grafos. Esta va a ser, pues, una estructura muy importante para muchos problemas que se puedan plantear, lo que la convierte en una candidata idónea para probar distintos lenguajes.

4.1. El problema

Uno de los problemas más conocidos, y que aún hoy en día se sigue estudiando, es el obtener el camino mínimo de un grafo. Para simplificar el problema, se suele asumir que el grafo es no dirigido, es decir, que las aristas no tienen sentido o dirección, y que los pesos de dichas aristas son enteros positivos. Edsger W. Dijkstra, famoso informático teórico, estudió este problema y en 1959 diseñó un algoritmo para resolverlo con un coste computacional excelente, sobre todo teniendo en cuenta su complejidad. [Dij59] Por tanto, la implementación de este algoritmo es una opción excelente para analizar el rendimiento de los dos lenguajes escogidos.

4.1. Los datos

Para obtener un análisis que se considere válido es imprescindible que los datos de entrada lo sean también. En este caso, los datos de entrada serán grafos y un vértice inicial. El vértice es indiferente a la hora de realizar el análisis, las únicas condiciones son que esté comprendido en el conjunto de vértices del grafo y que sea el mismo para todos los casos. En cambio, los grafos a introducir deben de suponer una muestra significativa y representativa de todos los posibles grafos que el algoritmo pueda recibir como entrada. En primer lugar, no todos los grafos poseen la misma densidad o número de aristas, por lo que será necesario generar grafos con distintas densidades. Para este caso se han escogido 3 divisiones o grupos: grafos dispersos, en los que el número de aristas es del 20% del total; grafos normales, con un 50% de las aristas; y grafos densos, en los que el número de aristas alcanza el 80%. En segundo lugar, el tamaño de los nodos es muy significativo a la hora de almacenar y recorrer el grafo, ya que para n vértices, el número de aristas podría llegar a ser n^2, que implica un coste cuadrático. Se ha considerado suficiente con establecer 4 divisiones o grupos en lo referente al tamaño del grafo para el análisis: 100, 500, 1000 y 5000 vértices. La conclusión que se deriva de ello es que se tendrán 12

grafos distintos, muestra representativa ya que explotan tanto el coste de almacenamiento como el coste computacional.

La generación de estos grafos ha sido de forma pseudo-aleatoria Se emplea este término debido a que al ser creados con un ordenador, la aleatoriedad como tal concepto no es posible, puesto que un ordenador es un sistema determinista. Aún así, los algoritmos aleatorios, más conocidos como *random*, han sido muy estudiados y hoy en día las implementaciones realizadas obtienen unos resultados excelentes. En este estudio se han utilizado dos funciones aleatorias para crear los grafos. La primera genera datos según una distribución uniforme entre 0 y 1, para permitir decidir, según el criterio de densidad de cada grafo, si la arista existe o no. En caso de no existir, el peso asignado será 0, mientras que, en caso de existir, se le asignará un peso positivo mayor que 0. Para obtener este peso de forma aleatoria se ha usado la segunda función *random*, que genera valores enteros según una distribución uniforme con un rango que abarca desde 1 hasta el número de vértices del grafo. Así, los 12 grafos obtenidos son independientes unos de otros, lo que reafirma el objetivo buscado de que los grafos supusieran una muestra representativa de la entrada.

5. La implementación

Se ha procurado que la implementación fuese lo más simple posible, tratando únicamente de aprovechar las propias fortalezas del lenguaje, como el tipo lista que contiene la librería estándar de Python o la creación de paquetes genéricos en Ada. El objetivo de este análisis es poder evaluar el propio lenguaje de programación, no lo bueno que pueda ser un programador para intentar optimizar hasta el más mínimo detalle. Por este mismo motivo, la compilación de Ada se ha realizado sin usar la opción de optimización que posee el propio compilador, dado que tampoco es objetivo de este análisis evaluar la eficiencia a la hora de optimizar código del compilador.

Aunque los resultados puedan verse influidos por las fortalezas y debilidades del algoritmo escogido, estos afectarán de igual manera a los dos lenguajes, por lo que este hecho no supondrá interferirá alguna en el análisis de los resultados.

Un problema que subyace ante el tratamiento de grafos es el cómo almacenarlos. Para llevar a cabo un estudio más completo de los lenguajes, se han escogido dos estructuras sencillas, ampliamente conocidas y estudiadas. La primera forma de almacenamiento es mediante una matriz de adyacencia. En ella se almacenan los pesos de todas las aristas del grafo, de forma que los dos índices que seleccionan cada elemento correspondan a los dos vértices que une dicha arista. Aquellos vértices no unidos por ninguna arista tendrán peso 0. Dado que el grafo es no dirigido, la matriz resultante será simétrica. La otra forma de almacenar el grafo será por listas de adyacencia. En esta estructura se tiene una lista por cada vértice del grafo, en la que se almacena únicamente la información de aquellos vértices para los que exista una arista que los una. En este caso dicha información será el peso de la arista y el vértice con el que lo une.

Como es lógico, cada una de las estructuras tiene una serie de características que afectarán a los resultados que se van a obtener. En el caso de la matriz de adyacencia, en general la cantidad de memoria requerida será superior que en las listas de adyacencia, aunque la información almacenada en la lista sea el doble (se tiene el vértice y el peso de la arista que los une). Es fácil deducir que mientras más disperso sea un grafo, menor será la información almacenada en las listas, mientras que en la matriz el tamaño permanecerá constante sea cual sea el número de aristas que haya. Mirando las características de estas estructuras en cuanto a la eficiencia o rendimiento computacional, la matriz obtiene unos resultados notablemente mejores que las listas. Esto se debe a que, para obtener el peso de una arista que une dos vértices i y j cualesquiera, en el caso de la matriz bastará

únicamente con acceder al elemento *(i, j)* de la matriz, mientras que en el caso de las listas, será necesario recorrerse toda la lista de uno de los vértices para ver si se encuentra el otro entre sus elementos almacenados.

Dado que el objetivo es medir la eficiencia de ambos lenguajes, es muy importante que la obtención del coste temporal sea correcta. Para conseguir este objetivo, se va a ejecutar la implementación del algoritmo de Dijkstra 100 veces en el cuerpo de un bucle iterativo. Se hará una captura del tiempo de CPU antes y después de dicho bucle, para así obtener el tiempo total de cómputo. Para ser más precisos, a continuación se ejecutará un nuevo bucle, esta vez vacío, con otras 100 iteraciones, y se hará una nueva captura de tiempos al inicio y al final del mismo. El objetivo de esta nueva medida es obtener el coste temporal que supone el propio bucle, obteniendo como resultado el coste exacto de ejecutar 100 veces la implementación realizada. Finalmente basta con dividir dicho cálculo por 100 para obtener como resultado el coste temporal de una única ejecución del algoritmo.

6. Resultados y análisis

Una vez establecido el enunciado y cómo se han resuelto los distintos puntos críticos, solo resta obtener los resultados de la ejecución y analizarlos.

6.1. Los resultados obtenidos

En las Tablas 1 y 2 se pueden ver los resultados obtenidos tras la ejecución, y en las Figuras 1, 2, 3 ,4 y 5 se muestran estos datos gráficamente para facilitar el análisis posterior.

Tabla 1. Resultados de la ejecución en Python (en segundos).

	Matriz			Listas		
Grafo	Disp.	Normal	Denso	Disp.	Normal	Denso
100	0,0033	0,0054	0,0066	0,0077	0,0374	0,0828
500	0,0781	0,1219	0,1636	0,6313	3,6349	9,2556
1000	0,3151	0,4901	0,6559	4,8253	29,299	74,057
5000	8,0775	12,388	16,623	582,20	3613,6	9227,1

Tabla 2. Resultados de la ejecución en Ada (en segundos).

	Matriz			Listas		
Grafo	Disp.	Normal	Denso	Disp.	Normal	Denso
100	0,0009	0,0023	0,0039	0,0009	0,0033	0,0067
500	0,0395	0,1374	0,2722	0,0587	0,2937	0,7364
1000	0,1818	0,8116	1,9557	0,4125	2,4220	6,3516
5000	21,943	125,68	274,84	53,201	314,34	753,02

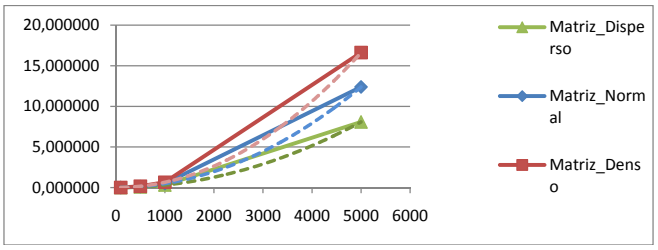

Figura 1. Líneas de tendencia para los resultados de Python con matriz de adyacencia.

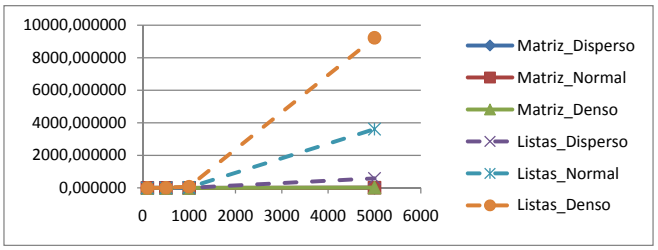

Figura 2. Representación gráfica de todos los resultados de Python.

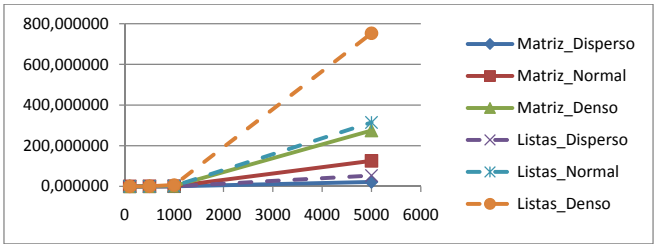

Figura 3. Representación gráfica de todos los resultados de Ada.

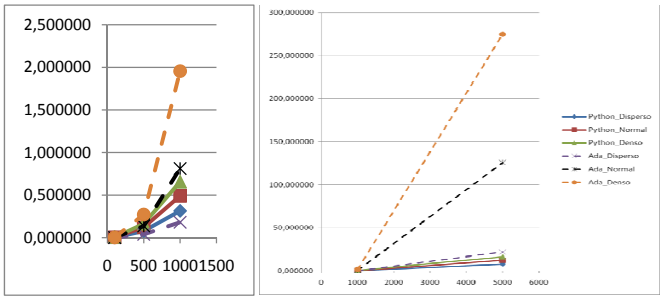

Figura 4. Comparativa de los resultados de Pyhton y Ada con matrices de adyacencia.

6.2. Análisis de los resultados

Para realizar un análisis adecuado de los datos obtenidos, en primer lugar se realizará un análisis exclusivo de cada lenguaje, observando su comportamiento para cada una de las estructuras implementadas. Y después, se realizará un análisis comparativo entre los dos lenguajes de programación, que es el objetivo que se planeaba al inicio de este documento.

Antes de centrarnos en los resultados obtenidos para cada una de las estructuras, es interesante parase un momento a observar los datos en pequeños conjuntos para obtener ciertas conclusiones. Si, por ejemplo, nos fijamos únicamente en los resultados obtenidos

para la matriz de adyacencia con Python, al representarlos en una grafica podemos observar cómo todos los puntos para cada tipo de grafo se pueden representar mediante una curva polinómica de grado 2. Este hecho queda reflejado en la *Figura 1*, donde se han representado dichos datos, así como sus líneas de tendencia. Esta conclusión confirma la validez del cálculo teórico del coste del algoritmo de Dijkstra. Además, este mismo hecho sucede para todos los grupos de datos equivalentes en las distintas estructuras y en ambos lenguajes.

Centrándonos en la *Figura 2*, lo primero que se ve es que los tiempos para la implementación en Python con matrices de adyacencia son mucho mejores que los tiempos obtenidos para las listas de adyacencia. Dadas las propiedades de las estructuras de almacenamiento utilizadas, los resultados son los esperados, pues el acceso a una matriz es mucho más rápido que el acceso a un elemento de una lista, ya que para llegar a él es necesario pasar por todos los elementos anteriores. Otro de los resultados esperados, es que los tiempos para el caso de tener como entrada un grafo disperso y almacenarlo en forma de listas de adyacencia, los tiempos son notoriamente mejores que para los otros dos tipos de grafos, ya que al tener un menor número de aristas, el número de elementos a recorrer es menor. Un resultado que no se esperaba y que en cambio se observa en los datos obtenidos, es el hecho de que todos los resultados con listas de adyacencia son peores que con las matrices, lo que no tiene sentido para el caso de listas de adyacencia con grafos dispersos, ya que los tiempos deberían ser mejores que los obtenidos para matrices con grafos densos, que supone una carga computacional mucho mayor.

Pasemos ahora a la *Figura 3*. En ella encontramos representados los datos para la implementación en Ada con ambas estructuras. Al igual que pasaba en el caso anterior, analizando los resultados de cada estructura por separado, se puede llegar a la conclusión de que los tiempos de computo obtenidos son los esperados debido a las propiedades inherentes de las propias estructuras utilizadas. Así pues, la matriz de adyacencia tiene un comportamiento muy bueno para todos los casos, mientras que las listas destacan únicamente cuando la entrada es un grafo disperso. Por el contrario, en este caso sí que los tiempos de las listas de adyacencia para grafos dispersos son mejores que los obtenidos al usar la matriz en grafos normales o densos, aunque sigue obteniendo mejores resultados la matriz de adyacencia para las mismas entradas. No es nada nuevo, ya que el coste en el acceso a la información de cada arista sigue siendo menor para las matrices.

Como último paso, queda realizar el análisis comparativo de ambos lenguajes. Dado que los resultados anteriores han dejado como claro ganador, en cuanto a eficiencia se trata, a la implementación basada en matrices, en la *Figura 4* se pueden ver los datos para ambos lenguajes de la implementación con esta estructura. Se ha realizado una división en 2 gráficas para poder ver con mayor claridad lo que sucede conforme aumenta el número de vértices. En la primera se observan los tiempos obtenidos hasta 1000 vértices, y en la segunda los tiempos de 1000 a 5000. Centrándonos en la gráfica de la izquierda, vamos a analizar los resultados para las distintas densidades de los grados. Tomando como entrada los grafos dispersos, queda claro que conforme aumenta el número de vértices, la implementación en Ada obtiene cada vez mejores tiempos que Python. En cambio, tanto para el caso de tener como entrada grafos normales o densos, puede verse como claramente Python obtiene mejores tiempos que Ada, sobre todo cuando el número de vértices pasa de 500 a 1000 (lo que supone pasar de 250000 datos a tener una matriz de un millón de elementos, es decir, cuadruplicar la información almacenada). Dado el crecimiento y comportamiento de ambas implementaciones, todo parece indicar que en el siguiente paso, al aumentar el tamaño de la entrada, los resultados seguirán la tendencia vista en los anteriores casos. Pero resulta realmente sorprendente el comprobar cómo esta suposición no es cierta. Al pasar a grafos

con 5000 vértices, el rendimiento de Ada cae en picado, siendo el mejor de los tiempos de Ada superior al mayor de los tiempos obtenidos con Python.

7. Conclusiones

De los resultados analizados en la *Figura 2* se deduce un hecho muy claro: la estructura de listas de adyacencia está contraindicada con el lenguaje de programación Python, pues los tiempos computaciones son mucho mayores que para la estructura matricial. Se puede plantear el usar esta estructura únicamente si el tiempo computacional no es un requisito indispensable en la solución del problema que nos planteen. Para casos en los que la cantidad de memoria es reducida y para grafos de entrada con un número de vértices pequeño, esta estructura puede considerarse adecuada, pero en general se optará siempre por una implementación con matriz de adyacencia.

Tras analizar los resultados obtenidos para la implementación en Ada, podemos concluir que en este caso será necesario un estudio de los requisitos del problema y del hardware donde vaya a funcionar nuestro programa. Si lo importante es el ahorro de memoria, no quedará más remedio que usar la implementación con listas de adyacencia, mientras que si se busca el mejor rendimiento, las matrices de adyacencia siguen siendo, sin duda, la mejor opción. Al igual que se ha comentado antes, si el problema deja la libertad de optar por cualquiera de las dos, sin duda la opción elegida será la matriz de adyacencia.

La conclusión que se puede extraer de la comparativa de los dos lenguajes es que, para poder elegir el que dará como resultado una solución mejor, será necesario realizar un estudio tanto de los requisitos del problema, como de las limitaciones en memoria, así como de los datos de entrada. De hecho, este último punto pasa a ser el más relevante debido a los resultados sorprendentes analizados en la *Figura 4*. El objetivo de este estudio será concretar, en la medida de lo posible, el rango de valores entre los cuales estarán los tamaños de los grafos que se inserten como entrada. También será importante, como ya ha quedado claro a lo largo de este documento, el poder definir el rango de densidades que tendrán dichos grafos. Por tanto, para grafos dispersos de hasta 1000 vértices, será mucho mejor utilizar una implementación en Ada basada en matrices de adyacencia, mientras que para el resto de casos será mejor usar la misma implementación con la misma base, pero en Python. Sin embargo, no hemos de olvidar nunca que si el sistema requiere que nuestro programa necesite poca memoria, Ada será el lenguaje de programación elegido, junto con las listas de adyacencia. No hemos de olvidar nunca otras propiedades ya comentadas en el apartado de características de estos lenguajes, pues a veces no nos interesará tanto la eficiencia o los recursos necesarios, si no que deberemos guiarnos por otros criterios que quedan ya fuera del objetivo de este documento.

Como tareas pendientes para terminar de completar este análisis, faltaría analizar por qué Ada tiene esa caída de rendimiento al pasar a grafos con un elevado número de vértices. Al hilo de este documento, resultaría interesante obtener los tiempos de la misma implementación de Python, pero esta vez con código compilado, para ver si realmente el código compilación obtiene mejores resultados que haciendo una ejecución interpretada. Además, otro caso interesante de análisis sería el intentar explotar al máximo las optimizaciones en cada uno de los lenguajes y volver a realizar un análisis comparativo de los nuevos resultados. Puede que, aunque Python obtenga mejores resultado para grafos con un número de vértices elevado, una implementación optimizada de Ada pudiese alterar esto resultados, dejando a Ada como el lenguaje de programación óptimo para aquellos problemas que tuviesen como entrada grafos.

Agradecimientos

A Elvira Mayordomo y Roberto Blanco, por enseñarme las maravillas de la bioinformática y abrirme la puerta a un mundo apasionante, a Ignacio Requeno, quien me ayudó programando los pilares de la implementación realizada con el lenguaje de programación Ada, y a Laura Espina, por darme el apoyo y animo que me hacía falta para terminar de dar forma a este proyecto.

REFERENCIAS

[Moo65] :
Gordon E. Moore. *Cramming more components onto integrated circuits.* Electronics, Volume 38, Number 8 (1965).

[Web1] :
http://news.cnet.com/2100-1001-984051.html
Noticias de la empresa *cnet*. [*Moore's Law to roll on for another decade*]

[Dij59] :
Edsger W. Dijkstra. *A note on two problems in connexions with graphs.* Numerische Mathematik 1, 269-271 (1959).

[Web3] :
http://netgocios.bligoo.com/content/view/37882/Que_es_un_paradigma.html
Blog titulado Netgocios. [*¿Qué es un paradigma?*]

[Web4] :
http://www.python.org
Página oficial del lenguaje de programación Python.

[Web5] :
http://en.wikipedia.org/wiki/Ada_(programming_language)
Enciclopedia virtual en ingles. [*Ada (programming language)]*

F

Tipos abstractos de datos

```
------------------------------------------------------------------------------
-- FILE:  doubly_linked_lists.ads
-- DESCRIPTION:  Tipo abstracto de dato Listas de doble enlace.
--
------------------------------------------------------------------------------
-- AUTHOR:  Álvarez Jarreta, Jorge    (jorgeal@unizar.es)
-- LAST MODIFICATION:  28/Ene/2009
-- LAST VERSION:  v 1.0
--
------------------------------------------------------------------------------
-- HISTORICAL REPORT:
--     Date               Author           Version        Modifications
--
--  28/Ene/2009    Álvarez Jarreta, Jorge    1.0     - Incorporación del tipo
--                                                     Cursor.
--                                                     Añadidos nuevos
--                                                     procedimientos y
--                                                     funciones.
--
--  30/Dic/2009    Álvarez Jarreta, Jorge    0.0     - Creación del fichero.
--
------------------------------------------------------------------------------

------------------------------------------------------------------------------
generic
------------------------------------------------------------------------------
type Item is private;
with function "=" (item1, item2 : in Item) return Boolean is <>;
------------------------------------------------------------------------------
package doubly_linked_lists is
------------------------------------------------------------------------------
------------------------------------------------------------------------------
-- Types :
------------------------------------------------------------------------------
type Cursor is private;
type DLList is private;

------------------------------------------------------------------------------
-- Constants :
------------------------------------------------------------------------------
No_Element : constant Cursor;

------------------------------------------------------------------------------
-- Procedures and functions of Cursor :
------------------------------------------------------------------------------
-----------------------------------------------------------------------------
function First (list : in DLList) return Cursor;
-----------------------------------------------------------------------------
-- Devuelve un cursor que apunta al primer elemento de list. En caso de ser
-- una lista vacía, devuelve No_Element.
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
function Last (list : in DLList) return Cursor;
-----------------------------------------------------------------------------
-- Devuelve un cursor que apunta al último elemento de list. En caso de ser
```

```
-- una lista vacía, devuelve No_Element.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Previous (cursor_list : in out Cursor);
-------------------------------------------------------------------------
-- Mueve cursor al elemento anterior al que apunta al ejecutar el
-- procedimiento. De ser el primer elemento de la lista, toma el valor
-- No_Element.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Next (cursor_list : in out Cursor);
-------------------------------------------------------------------------
-- Mueve cursor al elemento siguiente al que apunta al ejecutar el
-- procedimiento. De ser el último elemento de la lista, toma el valor
-- No_Element.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
function Get_Element (cursor_list : in Cursor) return Item;
-------------------------------------------------------------------------
-- Devuelve el item apuntado por cursor_list.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Modify (cursor_list : in Cursor; element : in Item);
-------------------------------------------------------------------------
-- Modifica el elemento apuntado por cursor_list, cambiando su contenido
-- por element.
-------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- Procedures and functions of DLList :
-----------------------------------------------------------------------------
-------------------------------------------------------------------------
function Create_DLList return DLList;
-------------------------------------------------------------------------
-- Devuelve una lista de doble enlace vacía.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Append (list : in out DLList; element : in Item);
-------------------------------------------------------------------------
-- Incluye el item element al final de la lista list.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Delete (list : in out DLList; cursor_list : in out Cursor);
-------------------------------------------------------------------------
-- Borra de la lista list el elemento apuntado por cursor_list, y pone el
-- de cursor_list a No_Element.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Delete_First (list : in out DLList; number : in Natural);
-------------------------------------------------------------------------
-- Borra los number primeros elementos de list.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
procedure Delete_Last (list : in out DLList; number : in Natural);
-------------------------------------------------------------------------
-- Borra los number últimos elementos de list.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
function First_Element (list : in DLList) return Item;
-------------------------------------------------------------------------
-- Devuelve el primer item de list.
-------------------------------------------------------------------------

-------------------------------------------------------------------------
function Last_Element (list : in DLList) return Item;
```

```
-------------------------------------------------------------------------------
-- Devuelve el último item de list.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Number_Of_Elements (list : in DLList) return Natural;
-------------------------------------------------------------------------------
-- Deuvelve el número de items de list.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Is_Empty (list : in DLList) return Boolean;
-------------------------------------------------------------------------------
-- Devuelve True si list está vacía, False en caso contrario.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Find (list : in DLList; element : in Item) return Cursor;
-------------------------------------------------------------------------------
-- Devuelve un cursor que apunta al item element de list. En caso de no
-- existir, se devolverá el valor No_Element.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Is_In (list : in DLList; element : in Item) return Boolean;
-------------------------------------------------------------------------------
-- Devuelve True si el item element está en list, False en caso contrario.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Copy (dest_list : in out DLList; source_list : in DLList);
-------------------------------------------------------------------------------
-- Realiza una copia exacta de source_list en dest_list, borrando su
-- contenido previo, si tenía.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Cut_And_Paste (dest_list, source_list : in out DLList);
-------------------------------------------------------------------------------
-- Realiza una copia exacta de source_list en dest_list, borrando el
-- contenido de ambas listas, si tenían.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Free (list : in out DLList);
-------------------------------------------------------------------------------
-- Libera la memoria ocupada por list.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
private
-------------------------------------------------------------------------------
type Node;
type Cursor is access Node;
No_Element : constant Cursor := null;
type Node is record
data : Item;
prev, next : Cursor;
end record;
type DLList is record
first, last : Cursor;
number_nodes : Natural;
end record;

-------------------------------------------------------------------------------
end doubly_linked_lists;
-------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
-- FILE:  dynamic_graphs.ads
-- DESCRIPTION:  Tipo abstracto de dato Grafos dinámicos.
--
------------------------------------------------------------------------------
-- AUTHOR:  Alvarez Jarreta, Jorge    (jorgeal@unizar.es)
-- LAST MODIFICATION:  14/Feb/2010
-- LAST VERSION:  v 0.0
--
------------------------------------------------------------------------------
-- HISTORICAL REPORT:
--     Date               Author           Version        Modifications
--
--  14/Feb/2010    Álvarez Jarreta, Jorge    0.0     - Creación del fichero.
--
------------------------------------------------------------------------------
with doubly_linked_lists;

------------------------------------------------------------------------------
generic
dyn_graph_size : Natural;
type NodeData is private;
with function "=" (node1, node2 : in NodeData) return Boolean is <>;
------------------------------------------------------------------------------
package dynamic_graphs is
------------------------------------------------------------------------------
------------------------------------------------------------------------------
-- Packages :
------------------------------------------------------------------------------
package Node_Lists is new doubly_linked_lists(NodeData, "=");
use Node_Lists;
package Subgraph_Lists is new doubly_linked_lists(Node_Lists.DLList, "=");
use Subgraph_Lists;

------------------------------------------------------------------------------
-- Types :
------------------------------------------------------------------------------
type DynamicGraph is private;

------------------------------------------------------------------------------
-- Procedures and functions :
------------------------------------------------------------------------------
------------------------------------------------------------------------------
procedure Create_Dynamic_Graph (dyn_graph : in out DynamicGraph;
node_list : in Node_Lists.DLList);
------------------------------------------------------------------------------
-- Pre: Length(node_list) = dyn_graph_size
-- Crea el grafo dinámico dyn_graph introduciendo todos los nodos en
-- node_list como nodos no conexos.
------------------------------------------------------------------------------

------------------------------------------------------------------------------
function Equal_Dynamic_Graphs (dyn_graph1, dyn_graph2 : in DynamicGraph)
return Boolean;
------------------------------------------------------------------------------
-- Devuelve True si dyn_graph1 y dyn_graph2 son iguales, False en caso
-- contrario.
------------------------------------------------------------------------------

------------------------------------------------------------------------------
procedure Set_Edge (dyn_graph : in out DynamicGraph;
node1, node2 : in NodeData; weight : in Natural);
------------------------------------------------------------------------------
-- Crea una arista en dyn_graph entre los nodos node1 y node2 y le asigna
-- el peso weight.
------------------------------------------------------------------------------

------------------------------------------------------------------------------
function Get_Edge (dyn_graph : in DynamicGraph; node1, node2 : in NodeData)
return Natural;
------------------------------------------------------------------------------
-- Devuelve el peso de la arista de dyn_graph que une node1 y node2.
------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
procedure Contract_Edge (dyn_graph : in out DynamicGraph;
node1, node2 : in NodeData);
-------------------------------------------------------------------------------
-- Borra la arista de dyn_graph que une los nodos node1 y node2, uniendo
-- dichos nodos en uno solo, que mantendrá las demás aristas que tenían
-- ambos.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Delete_Edge (dyn_graph : in out DynamicGraph;
node1, node2 : in NodeData);
-------------------------------------------------------------------------------
-- Borra la arista de dyn_graph que une los nodos node1 y node2.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Is_Disconnected (dyn_graph : in DynamicGraph) return Boolean;
-------------------------------------------------------------------------------
-- Devuelve True si dyn_graph es no conexo, False en caso contrario.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Get_Disconnected_Graphs_Nodes (dyn_graph : in DynamicGraph)
return Subgraph_Lists.DLList;
-------------------------------------------------------------------------------
-- Devuelve una lista donde cada elemento es una lista de los nodos que
-- componen cada uno de los grafos conexos que conforman dyn_graph.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Minimum_Cut_Set (dyn_graph : in DynamicGraph)
return Node_Lists.DLList;
-------------------------------------------------------------------------------
-- Devuelve una lista con el conjunto de nodos de corte mínimo de dyn_graph.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Free_Dynamic_Graph (dyn_graph : in out DynamicGraph);
-------------------------------------------------------------------------------
-- Libera la memoria ocupada por dyn_graph.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
private
-------------------------------------------------------------------------------
-- Types :
-------------------------------------------------------------------------------
type IndexArray is Array (1 .. dyn_graph_size) of Node_Lists.DLList;
type IndexPointer is access IndexArray;
type Graph is Array (1 .. dyn_graph_size, 1 .. dyn_graph_size) of Natural;
type GraphPointer is access Graph;
type DynamicGraph is record
indexes : IndexPointer;
matrix : GraphPointer;
end record;

-------------------------------------------------------------------------------
end dynamic_graphs;
-------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
-- FILE:  phylogenetic_trees.ads
-- DESCRIPTION:  Tipo abstracto de dato Árbol filogenético.
--
------------------------------------------------------------------------------
-- AUTHOR:  Alvarez Jarreta, Jorge    (jorgeal@unizar.es)
-- LAST MODIFICATION:  25/Feb/2010
-- LAST VERSION:  v 0.1
--
------------------------------------------------------------------------------
-- HISTORICAL REPORT:
--     Date               Author           Version        Modifications
--
--  25/Feb/2010    Alvarez Jarreta, Jorge    0.1     - Cambio de la estructura
--                                                     de datos para mejorar
el
--                                                     rendimiento.
--
--  19/Feb/2010    Alvarez Jarreta, Jorge    0.0     - Creación del fichero.
--
------------------------------------------------------------------------------
with doubly_linked_lists;
with ada.text_io, ada.strings.unbounded;
use  ada.text_io, ada.strings.unbounded;

------------------------------------------------------------------------------
generic
type LeafData is private;
with function "=" (leaf1, leaf2 : in LeafData) return Boolean is <>;
with function To_LeafData (leaf : in Unbounded_String) return LeafData;
with function To_Unbounded_String (leaf : in LeafData)
return Unbounded_String;
------------------------------------------------------------------------------
package phylogenetic_trees is
------------------------------------------------------------------------------
------------------------------------------------------------------------------
-- Types :
------------------------------------------------------------------------------
type PhylogeneticTree is private;

------------------------------------------------------------------------------
-- Packages :
------------------------------------------------------------------------------
package Tree_Leaves_Lists is new doubly_linked_lists(LeafData, "=");
use Tree_Leaves_Lists;

------------------------------------------------------------------------------
-- Procedures and functions :
------------------------------------------------------------------------------
-----------------------------------------------------------------------------
function Create_Rooted_Phylogenetic_Tree return PhylogeneticTree;
-----------------------------------------------------------------------------
-- Devuelve un árbol filogenético vacío.
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
function Is_Empty_Phylogenetic_Tree (tree : in PhylogeneticTree)
return Boolean;
-----------------------------------------------------------------------------
-- Devuelve True si tree esta vacío, False en caso contrario.
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
function "=" (tree1, tree2 : in PhylogeneticTree) return Boolean;
-----------------------------------------------------------------------------
-- Devuelve True si tree1 y tree2 son iguales (mismas hojas y mismas
-- relaciones), False en caso contrario.
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
procedure Copy_Phylogenetic_Tree (tree_in : in PhylogeneticTree;
tree_out : out PhylogeneticTree);
-----------------------------------------------------------------------------
-- Copia todo el contenido de tree_in en tree_out.
```

```
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Add_Leaf (tree : in out PhylogeneticTree; leaf : in LeafData);
-------------------------------------------------------------------------------
-- Añade la hoja leaf al árbol tree, como hija de la raíz del árbol.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Add_SubTree (tree, subtree : in out PhylogeneticTree);
-------------------------------------------------------------------------------
-- Añade el subárbol subtree al árbol tree, situando la raíz de subtree
-- como hijo de la raíz de tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Set_Weight (tree : in out PhylogeneticTree; weight : in Natural);
-------------------------------------------------------------------------------
-- Asigna el peso weight al árbol tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Get_Weight (tree : in PhylogeneticTree) return Natural;
-------------------------------------------------------------------------------
-- Devuelve el peso de tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function Get_Leaves_List (tree : in PhylogeneticTree)
return Tree_Leaves_Lists.DLList;
-------------------------------------------------------------------------------
-- Devuelve una lista con las hojas que forman parte de tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
function In_Proper_Cluster (tree : in PhylogeneticTree;
leaf1, leaf2 : in LeafData) return Boolean;
-------------------------------------------------------------------------------
-- Devuelve True si leaf1 y leaf2 están en el mismo cluster de tree, False
-- en caso contrario. Que dos hojas estén en el mismo cluster implica que
-- exista un nodo que las tenga a ambas como descendientes y que no sea la
-- raíz.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Get_Minimum_Subtree (tree : in out PhylogeneticTree;
leaves : in Tree_Leaves_Lists.DLList);
-------------------------------------------------------------------------------
-- Obtiene el subárbol mínimo de tree que contiene todas las hojas de la
-- lista leaves.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Free_Phylogenetic_Tree (tree : in out PhylogeneticTree);
-------------------------------------------------------------------------------
-- Libera la memoria ocupada por tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Read_Phylogenetic_Tree (file : in out File_Type;
tree : out PhylogeneticTree);
-------------------------------------------------------------------------------
-- Lee el árbol filogenético (en formato Newick) de file y crea el árbol
-- filogenético tree.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
procedure Write_Phylogenetic_Tree (file : in out File_Type;
tree : in PhylogeneticTree);
-------------------------------------------------------------------------------
-- Traduce el árbol tree a formato Newick y lo escribe en file.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
```

```
private
-------------------------------------------------------------------------------
package Proper_Clusters is new doubly_linked_lists(Natural, "=");
use Proper_Clusters;
type TreeLeaf is record
data : LeafData;
proper_cluster : Proper_Clusters.DLList :=
Proper_Clusters.Create_DLList;
end record;
type TreeLeafPointer is access all TreeLeaf;
-------------------------------------------------------------------------------
function "=" (leaf1, leaf2 : in TreeLeaf) return Boolean;
-------------------------------------------------------------------------------
-- Devuelve True si leaf1 y leaf2 son iguales (tanto el valor de data como
-- las listas proper_cluster), False en caso contrario.
-------------------------------------------------------------------------------
package Trees_Skeleton is new doubly_linked_lists(TreeLeaf, "=");
use Trees_Skeleton;
type PhylogeneticTree is record
root : Natural := 0;
skeleton : Trees_Skeleton.DLList := Trees_Skeleton.Create_DLList;
weight : Natural := 0;
end record;

-------------------------------------------------------------------------------
end phylogenetic_trees;
-------------------------------------------------------------------------------
```

G

Árboles filogenéticos para las pruebas del algoritmo de superárboles de corte mínimo

Prueba 1

Árboles de entrada:

(((A,B),((C,D),(E,F)),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));
(((A,B),((C,D,E),F),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));
(((A,B),(C,D,E,F),(G,H,I)),(((J,K),(L,M)),(N,O,(P,Q))));

Árbol de salida:

(((A,B),(((C,D),E),F),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));

Prueba 2

Árboles de entrada:

(((A,B),((C,D),(E,F)),(G,(H,I,S))),(((J,K),(L,M)),(R,(N,O),(P,Q))));
(((A,B),((C,D,E),F),S,(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));
(((A,B),(C,D,E,F),(G,H,I)),(((J,T,K),(L,M)),(N,O,(P,Q))));

Árbol de salida:

(((A,B),(((C,D),E),F),((G,(H,I)),S)),(((J,K,T),(L,M)),(R,(N,O),(P
,Q))));

Prueba 3

Árboles de entrada:

(((A,B),((C,D),(E,F)),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));
(((a,b),((c,d),(e,f)),(g,(h,i))),(((j,k),(l,m)),((n,o),(p,q))));

Árbol de salida:

(((A,B),((C,D),(E,F)),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))),
((a,b),((c,d),(e,f)),(g,(h,i))),(((j,k),(l,m)),((n,o),(p,q))));

Prueba 4

Árboles de entrada:

(((A,B),((C,D),(E,F)),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))));
(((a,b),(c,d,e)),((f,g),(h,i),(j,k),(l,m)),((n,o,p),(q,r)));

Árbol de salida:

(((A,B),((C,D),(E,F)),(G,(H,I))),(((J,K),(L,M)),((N,O),(P,Q))),
((a,b), (c,d,e)),((f,g),(h,i),(j,k),(l,m)),((n,o,p),(q,r)));

H

Manual de usuario: *Programa de construcción de superárboles de corte mínimo*

Para hacer uso del algoritmo de construcción de superárboles de corte mínimo solo hay que lanzar el ejecutable *mcst*. El programa irá solicitando la información que requiere para poder realizar la ejecución del método. Un vez finalizado, informará de este hecho al usuario con un mensaje por pantalla y terminará su ejecución.

Como datos de entrada, el programa solicita uno o más ficheros con los árboles filogenéticos a procesar y el nombre que se le desea dar al fichero donde se guardará el superárbol resultado de aplicar el algoritmo. En los ficheros de entrada se deben encontrar los árboles filogenéticos en formato Newick [3]. En un fichero puede haber más de un árbol, siempre que estos se encuentren en líneas distintas (el ";" final, pese a ser indicador de fin del árbol, no se ha utilizado para diferenciar entre árboles dentro del mismo fichero). En el fichero de salida se podrá encontrar el superárbol, también en formato Newick.

Bibliografía

[1] http://atgc.lirmm.fr/phyml/. Web site del software Phyml.

[2] http://evolution.genetics.washington.edu/phylip.html. Web site del paquete Phylip.

[3] http://evolution.genetics.washington.edu/phylip/newicktree.html. Web site donde se explica el formato Newick para árboles filogenéticos.

[4] http://www.cs.wisc.edu/condor/manual/. Web site del manual de Condor.

[5] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertran Ludäscher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004.

[6] Roberto Blanco. Definición y prototipo de herramienta de análisis filogenético para el ADN mitocondrial humano. Master's thesis, Centro Politécnico Superior, Universidad de Zaragoza, 2008.

[7] Roberto Blanco. Computational phylogenetics for human mitochondrial DNA. Master's thesis, Centro Politécnico Superior, Universidad de Zaragoza, 2009.

[8] Roberto Blanco and Elvira Mayordomo. ZARAMIT: a system for the evolutionary study of human mitochondrial DNA. In *IWANN 2009, Part II*, volume 5518 of *Lecture Notes in Computer Science*, pages 1139–1142, 2009.

[9] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. *Workflows for e-Science*, chapter Workflow management in Condor, pages 357–375. Springer, 2006.

[10] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, V1(1):25–39, 2003.

[11] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[12] Corinna Herrnstadt, Joanna L. Elson, Eoin Fahy, Gwen Preston, Douglass M. Turnbull, Christen Anderson, Soumitra S. Ghosh, Jerrold M. Olefsky, M. Flint Beal, Robert E. Davis, and Neil Howell. Reduced-median-network analysis of complete mitochondrial DNA coding-region sequences for the major African, Asian, and European haplogroups. *American Journal of Human Genetics*, 70(5):1152–1171, 2002.

[13] Sergei L. Kosakovsky Pond and Spencer V. Muse. *Statistical Methods in Molecular Evolution*, chapter HyPhy: hypothesis testing using phylogenies, pages 125–182. Springer, 2005.

[14] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[15] Nicholas M. Luscombe, Dov Greenbaum, and Mark Gerstein. What is bioinformatics? an introduction and overview. In *Yearbook of Medical Informatics 2001*, 2001.

[16] Wayne P. Maddison. Gene trees in species trees. *Systems Biology*, 46(3):523–536, 1997.

[17] Timothy McPhillips, Shawn Bowers, and Bertram Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *International Workshop on Data Integration in the Life Sciences*, 2006.

[18] Timothy M. McPhillips. Pipelined scientific workflows for inferring evolutionary relationships. Technical report, National Diversity Discovery Project, 2005.

[19] Timothy M. McPhillips, Shawn Bowers, and Bertram Ludäscher. Data and workflow management for the atol program. Poster, Genome Center Symposium, 2006.

[20] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[21] Roderic D. M. Page. Modified mincut supertrees. In *Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 537–552, 2002.

[22] Helen Piontkivska. Efficiencies of maximum likelihood methods of phylogenetic inferences when different substitution models are used. *Molecular Phylogenetics and Evolution*, 31:865–873, 2004.

[23] Andrzej Polanski and Marek Kimmel. *Bioinformatics*. Springer, 2007.

[24] David Posada. *The phylogenetic handbook*, chapter Selecting models of evolution, pages 256–282. Cambridge University Press, 2003.

[25] David Posada. jModelTest: phylogenetic model averaging. *Molecular Biology and Evolution*, 25(7):1253–1256, 2008.

[26] David Posada and Thomas R. Buckley. Model selection and model averaging in phylogenetics: Advantages of akaike information criterion and bayesian approaches over likelihood ratio tests. *Systematic Biology*, 53:793–808, 2004.

[27] David Posada and Keith A. Crandall. MODELTEST: testing the model of DNA substitution. *Bioinformatics*, 14(9):817–818, 1998.

[28] David Posada and Keith A. Crandall. Selecting models of nucleotide substitution: An application to human immunodeficiency virus 1 (hiv-1). *Molecular Biology and Evolution*, 18:897–906, 2001.

[29] David Posada and Keith A. Crandall. Selecting the best-fit model of nucleotide substitution. *Systematic Biology*, 50(4):580–601, 2001.

[30] Manfred Reichert and Peter Dadam. ADEPTflex: supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[31] Martin B. Richards, Vincent A. Macaulay, Hans-Jürgen Bandelt, and Bryan C. Sykes. Phylogeography of mitochondrial DNA in western Europe. *Annals of Human Genetics*, 62(3):241–260, 1998.

[32] Eduardo Ruiz-Pesini, Dan Mishmar, Martin Brandon, Vincent Procaccio, and Douglas C. Wallace. Effects of purifying and adaptive selection on regional variation in human mtdna. *Science*, 303:223–226, 2004.

[33] Andrey Rzhetsky and Tatyana Sitnikova. When is it safe to use an oversimplified substitution model in tree-making? *Molecular Biology and Evolution*, 13:1255–1265, 1996.

[34] Antonio Salas, Victor Lareu, Francesc Calafell, Jaume Bertranpetit, and Ángel Carracedo. mtdna hypervariable region ii (hvii) sequences in human evolution studies. *European Journal of Human Genetics*, 8:964–974, 2000.

[35] Michael J. Sanderson, Andy Purvis, and Chris Henze. Phylogenetic supertrees: assembling the trees of life. *Trends in Ecology and Evolution*, 13(3):105–109, 1998.

[36] Charles Semple and Mike Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105:147–158, 2000.

[37] Dadabhai T. Singh, Rahul Trehan, Bertil Schmidt, and Timo Bretschneider. Comparative phyloinformatics of virus genes at micro and macro levels in a distributed computing environment. *BMC Bioinformatics*, 9(Suppl. 1):S23, 2008.

[38] Mike Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9(1):91–116, 1992.

[39] Mike Steel. The maximum likelihood point for a phylogenetic tree is not unique. *Systematic Biology*, 43:560–564, 1994.

[40] Mike Steel. Parsimony, likelihood, and the role of models in molecular phylogenetics. *Molecular Biology and Evolution*, 17:839–850, 2000.

[41] Mike Steel, Andreas W. M. Dress, and Sebastian Böcker. Simple but fundamental limitations on supertree and consensus tree methods. *Systematic Biology*, 49(2):363–368, 2000.

[42] Korbinian Strimmer. *Maximum Likelihood Methods in Molecular Phylogenetics*. PhD thesis, Ludwig-Maximilians-Universit ? t Munchen, 1997.

[43] Jack Sullivan and Paul Joyce. Model selection in phylogenetics. *Annual Review of Ecology, Evolution, and Systematics*, 36:445–466, 2005.

[44] Antonio Torroni, Alessandro Achilli, Vincent Macaulay, Martin Richards, and Hans-Jürgen Bandelt. Harvesting the fruit of the human mtDNA tree. *Trends in Genetics*, 22(6):339–345, 2006.

[45] Iván D. Traveso. Desarrollo e implementación de un entorno de análisis filogenético basado en la clasificación de haplogrupos mitocondriales y métodos de superárboles. Master's thesis, Centro Politécnico Superior, Universidad de Zaragoza, 2009.

[46] Pablo Urcola. Algoritmos de compresión para secuencias biológicas y su aplicación en árboles filogénicos construidos a partir de adn mitocondrial. Master's thesis, Universidad de Zaragoza, 2006.

[47] W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[48] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[49] Ziheng Yang. How often do wrong models produce better phylogenies? *Molecular Biology and Evolution*, 14:105–108, 1997.