Universidad Nacional Autónoma de México

Facultad de Ciencias

La secuenciación del ADN por medio del problema del agente viajero

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Matemáticas Aplicadas

P R E S E N T A :

José Fernando Méndez Torres

TUTOR

M. en I.O. María del Carmen Hernández Ayuso

Ciudad Universitaria, Cd. Mx., 2022

Prefacio

Durante los meses de septiembre y octubre de 2019, mi asesora y yo buscábamos un problema de índole biológica que fuera resuelto con metodologías de investigación de operaciones para la elaboración de mi tesis. Fue una grata sorpresa cuando encontré el libro *Integer Linear Programming in Computational and Systems Biology* de Gusfield [1]. Al mostrárselo a Carmen, ella le dio una hojeada rápida al índice y me señaló el capítulo 9, titulado *Traveling Salesman Problems In Genomics* comentando "Este tema me gusta para tu tesis. ¿Te parece?".

Durante febrero 2021 comencé a indagar sobre el tema principal de la tesis y pese a las dificultades que se presentaron con el paso de los meses, el día 19 de abril de 2021 presenté mi primera propuesta final. Ya con los comentarios de mi asesora y mis sinodales, en enero de 2022 el trabajo está finalizado. Esta sección es la última que he añadido y considero que es la más importante, ya que este espacio contiene mis agradecimientos hacia las personas que me apoyaron durante la carrera y en la elaboración de la tesis.

Respecto a la carrera

Agradezco a Lourdes Velasco y al comité de a los miembros de la comisión para la creación de la licenciatura, gracias a ustedes tuve la oportunidad de entrar a una carrera que me retó y me satisfizo. Además, agradezco a Roberto Montoya y Miguel Astorga, los profesores que me mostraron la belleza que se esconde detrás de los métodos matemáticos, por motivarme estudiar en esta área y guiarme en la búsqueda de la carrera.

Quiero agradecerle a mis amigos, Aldo, Agustín, Aníbal, Bruno, Carolina, Emilio, Iker, Infante, Luis, Luz, Mauricio, Michel, Montserrat, Roberto, Silvana, Victoria y Zamora por regalarme momentos tan extraordinarios antes y durante la carrera, así como estar para mí cuando más los necesité. Agradezco a mis amigos y compañeros de la facultad, Erick "Panchito" Navarrete, Sergio Fernández, Diego Lecuona, Luis García, David Vargas, Melissa Miranda, Ixchel Sampayo, Rafael Flores, Fernando Trejo y Karen Ochoa, por hacerme disfrutar las largas horas de desvelo en la realización de trabajos. Sin ayuda de ustedes no hubiera terminado la carrera en tiempo y forma.

A Elisa Domínguez, por abrirme las puertas a su laboratorio de Biología en Sistemas y a su cubículo, así como por introducirme al área de biología en sistemas. También a Dalia, Eliezer y Adán, mis ex-compañeros de laboratorio, por las discusiones científicas y las largas pláticas.

A mis profesores de Análisis Numérico y Probabilidad, Guilmer González y Begoña Fernández, por darme la oportunidad de empezar mi carrera como académico de ser su ayudante.

Finalmente quiero dar un agradecimiento a Fernanda Sánchez, con quien tuve el placer de ser ayudante, además de estar al pendiente de mí en la carrera y siempre tenía tiempo para platicar, tomar un café e ir al billar.

Respecto a la tesis

Agradezco a mi tutora, Carmen Hernández, por su tiempo, paciencia y dedicación en mi trabajo. Carmen, espero hayas disfrutado la elaboración de este trabajo tanto como yo.

A mis sinodales, Elisa Domínguez, Claudia López, Fernanda Sánchez y Ana Lilia Anaya por sus valiosos comentarios, sugerencias, observaciones y tiempo.

A Imanol Nuñez, Laura López, Eliezer Flores, Mauricio Jiménez, David Moreno y Brenda Ramírez, quienes me asesoraron en cuestiones técnicas y de escritura en partes fundamentales del texto, además de brindarme comentarios constructivos.

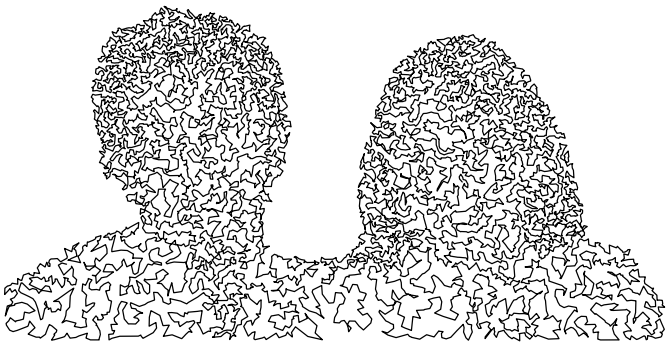
A Silvana López por acompañarme durante largas horas, escucharme divagar sobre ideas y darme su opinión con respecto a varios detalles de la tesis.

Nota del autor

Este trabajo fue elaborado con la paleta *Color Blind Safe* de IBM. Para la realización de la tesis se utilizaron los softwares LaTeX, Inkscape y Python3.

A mis padres,
Laura y Fernando,
quienes me formaron
y han sido mi mayor motivación.

Mamá, papá, quiero decirles que
esta carrera me ha dado
satisfacciones que nunca imaginé
y me promete un buen futuro.
Hoy me siento dichoso y afortunado
de compartir este logro con ustedes.
Espero estén orgullosos de mí.
Los amo.



Índice general

1. Secuenciación del ADN **5**
1.1. Principios de biología molecular 5
1.1.1. Enlaces químicos y moléculas orgánicas 5
1.1.2. Proteínas 7
1.1.3. Enzimas 9
1.2. Los ácidos nucléicos y el dogma central 10
1.2.1. Nucleótidos 11
1.2.2. Ácidos desoxirribonucleicos 11
1.2.3. Ácidos ribonucleicos y el dogma central de la biología molecular . 14
1.3. Secuenciación de lecturas 17
1.3.1. Secuenciación Sanger 18
1.3.2. Secuenciación de Illumina 18

2. El Problema del Agente Viajero **21**
2.1. Conceptos Preliminares 21
2.1.1. Planteamiento general 23
2.1.2. El problema del mensajero 25
2.2. El Problema de Asignación 26
2.2.1. Planteamiento general 26
2.2.2. El algoritmo húngaro 28
2.3. Algoritmo de ramificación y acotamiento 38
2.3.1. Un ejemplo de programación entera 38
2.3.2. Detección de circuitos 42
2.3.3. Ramificación y acotamiento para el PAV 44

3. Ordenamiento de fragmentos **51**
3.1. Errores e interpretación de datos computacionales 52
3.1.1. Errores de secuenciación 52
3.1.2. Formato FASTQ 54
3.2. Adaptación del PAV al ordenamiento de lecturas 55
3.2.1. Prefijos, sufijos y traslapes 55
3.2.2. Red de traslapes 56
3.3. Ensamblaje con errores de secuenciación 60
3.3.1. Manejo de errores 61
3.3.2. Una métrica de errores 63
3.3.3. Red de traslapes aproximados 65

4. Complejidad y heurísticas **69**
4.1. Teoría de complejidad y el PAV 69
4.1.1. Máquinas de Turing 69
4.1.2. Problemas P y NP 73
4.1.3. Problemas NP-Completos y NP-Duros 75
4.2. Una heurística para el PAV 83
4.2.1. Las heurísticas como alternativas de optimización 83
4.2.2. Algoritmos genéticos 85

5. Aplicación al fago λ **89**
5.1. Generación de datos *in silico* y la matriz de costos 89
5.1.1. Creación del archivo FASTQ 90
5.1.2. Limpieza de datos y matriz de costos 91
5.2. Secuenciación del genoma *Escherichia phage Lambda* 92
5.2.1. Algoritmo de ramificación y acotamiento 92
5.2.2. Algoritmo genético 93

A. Elementos de Investigación de Operaciones **97**
A.1. Gráficas y redes 97
A.2. Programación matemática 99

B. Pseudocódigos y diagramas de flujo **101**
B.1. Capítulo 2 101
B.2. Capítulo 3 107
B.3. Capítulo 4 109

Índice de figuras

1.1. Representación de enlaces covalentes. 6
1.2. Puente de hidrógeno entre dos moléculas de agua. 6
1.3. Estructura básica de los aminoácidos. 7
1.4. Estructura de un dipéptido. 8
1.5. Diferentes estructuras de la hemoglobina. 9
1.6. Pasos de una reacción enzimática con la relación cerradura–llave. 10
1.7. Molécula de ADN. 12
1.8. Replicación de ADN. 13
1.9. Transcripción de un gen. 15
1.10. Maduración del ARNm. 15
1.11. Traducción del ARNm a proteína. 16
1.12. El dogma central de la biología molecular. 17
1.13. Secuenciación Sanger. 19
1.14. Amplificación de la lectura. 19
1.15. Un ciclo de la SBS. 20

2.1. Gráfica de ciudades y rutas aéreas. 22
2.2. No necesariamente $c_{i,j} = c_{j,i}$. 25
2.3. Gráfica de las tareas y los integrantes del equipo. 28
2.4. Notación para el algoritmo húngaro. 33
2.5. Esquema para cada subiteración. 34
2.6. Gráfica del ejemplo 2.3. 39
2.7. El árbol del algoritmo de ramificación y acotamiento para el ejemplo 2.3. 41
2.8. Las regiones definidas por los nodos terminales. 42
2.9. El algoritmo de ramificación y acotamiento para el ejemplo 2.1. 50

3.1. Secuenciación por escopeta. 51
3.2. Errores de agrupamiento, fase y atenuación. 53
3.3. Probabilidad de que una base sea incorrecta en función de su calidad. . . 54
3.4. Costo de las cadenas del ejemplo 3.4. 57
3.5. Red de traslapes. 59
3.6. Trie del ejemplo 3.9. 62
3.7. Costo de las lecturas del ejemplo 3.13. 66
3.8. Red de traslapes. 67

4.1. Máquina de Turing de configuración $00q_1$ a $0q_{13}1$. 70
4.2. Diagrama de flujo de la función de transición δ_1. 71
4.3. Ramificación del algoritmo no determinista del ejemplo 4.2. 72

4.4. Pasos de ejecución para máquinas deterministas y no deterministas. . . . 74
4.5. Concepto de reducción polinomial. 76

5.1. Generación de la matriz de costos. 90
5.2. Cortes aleatorios. 91
5.3. Obtención de la secuencia con el algoritmo de RyA. 93
5.4. Obtención de la secuencia con el algoritmo genético. 94
5.5. La mejor ejecución del algoritmo genético. 94

B.1. Diagrama de flujo del algoritmo de ramificación y acotamiento. 106
B.2. Diagrama de flujo del algoritmo genético. 112

Índice de tablas

1.1. Diferencias entre el ADN y el ARN. 14

2.1. El tiempo estimado de viaje entre una ciudad y otra en minutos. 22
2.2. Los minutos que tardarían en resolver cada ejercicio. 27
2.3. Algoritmo de preprocesamiento para el Ejemplo 2.2. 32
2.4. Ejemplo de detección de componentes conexas. 44

3.1. Detección de traslape. 58
3.2. Sufijos de las lecturas del ejemplo 3.9 61
3.3. Calidad de los traslapes aproximados entre λ_1 y λ_2. 66

Introducción

El ADN es la molécula que contiene las instrucciones para generar proteínas, las cuales se encargan del crecimiento, desarrollo, funcionalidad y reproducción de los seres vivos y es representada mediante una **secuencia** o sucesión de cuatro letras, donde cada letra corresponde a un compuesto molde llamado base nitrogenada [2]. Se denomina **secuenciación** a la determinación del orden de las cuatro diferentes bases de alguna de estas moléculas, es decir, la determinación de la secuencia [3]. Para ello, existen múltiples técnicas que han sido desarrolladas a lo largo de las últimas décadas. La cuestión es que dichos métodos sólo pueden obtener secuencias cortas de ADN. Conocer el orden de las bases en estos compuestos nos permite saber cómo funcionan los organismos a nivel molecular, por lo que es de gran interés para las ciencias biológicas y de la salud.

Si poseemos la secuencia de algún individuo de cierta especie, obtener la de cualquier otro miembro resulta más fácil, ya que se tiene una referencia para las demás. Como menciona Langmead [4], los métodos de secuenciación nos brindan las piezas de un rompecabezas y, por ende, la molécula de referencia funge como la imagen en la caja que nos indica cómo armarlo. El problema surge en cómo obtener dicha secuencia, es decir ¿cómo obtenemos la imagen de la caja? Una técnica para obtenerla es a través del proceso de **secuenciación por escopeta**, el cual consiste de un procedimiento experimental y un análisis informático [1, pág. 159].

A partir del una molécula de ADN, podemos obtener fragmentos de una longitud secuenciable que posteriormente nos permitirá conocer cómo ensamblar la molécula original. Incluso dados dos fragmentos de ADN, pertenecientes al mismo ADN, podemos ver qué tan parecidos son a partir de los traslapes entre ellos [1, pág. 161].

```
GAGACGCCGTTTGGCCTCACGCCGGA
      CCGTTTGGCCTCACGCCGGATGAC
```

A mayor traslape, existe mayor evidencia del orden entre dos fragmentos, sin embargo no podemos limitarnos a dos traslapes. Considerando n fragmentos obtenidos de una molécula, debemos tomar en cuenta las $n!$ posibles formas de ordenarlas, donde aquella que nos dé la cadena de caracteres de menor longitud podría ser la mejor candidata al ADN original.

```
GAGACGCCGTTTGGCCTCAAATGGACGCCGGA
        GTTTGGCCTCAAATGGACGCCGGATGACCCCT
                    GGACGCCGGATGACCCCTCCAGCGTGTTT
                                TGACCCCTCCAGCGTGTTTTATCTCTGCGA
-----------------------------------------------------------
GAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA
```

Una forma de modelar lo anterior, para obtener la molécula original, es a través de un problema de optimización de redes llamado el **problema del mensajero (PM)**. El planteamiento original nos dice que un mensajero debe de entregar cartas a n direcciones diferentes en el menor costo, pasando por cada dirección una única vez. En nuestro caso, las secuencias de fragmentos corresponden a las direcciones y el costo de ir de un fragmento a otro está en función del traslape.

Es posible transformar el PM al **problema del agente viajero (PAV)**, el cual consiste en que una persona debe recorrer un listado de ciudades al menor costo, pasando una única vez por cada una de ellas y regresando a la ciudad de origen. La ventaja de utilizar el planteamiento del PAV es que el problema ha sido altamente estudiado desde la década de 1930 hasta la fecha, por lo que se conoce una gran variedad de formas de abordarlo.

El objetivo principal de este trabajo es modelar matemáticamente el ordenamiento de secuencias de ADN como un problema del agente viajero así como proponer métodos para resolverlo y programarlos en `Python3`. Como objetivo adicional, este trabajo aporta material didáctico para algunos cursos como programación entera y teoría de redes, ya que incluye el desarrollo de temas concernientes a dichos cursos, así como el software.

Este texto está dividido en 5 capítulos los cuales contienen lo siguiente:

En el primer capítulo veremos algunos conceptos básicos de biología molecular, como lo son los enlaces químicos y las moléculas orgánicas para poder describir a las proteínas, las enzimas y al ADN, las cuales son moléculas esenciales para el proceso experimental de la secuenciación. También veremos el dogma central de la biología molecular, el cual es una simplificación del flujo de información de cómo se sintetizan las proteínas a partir del ADN dentro de las células. Lo anterior será útil para entender algunos procesos químicos que se utilizan en los métodos de secuenciación. En este mismo capítulo se presentan dos métodos utilizados para conocer la secuencia de pequeñas moléculas de ADN, la secuenciación Sanger y la secuenciación de Illumina. Para los fines de esta tesis, es de mayor interés conocer el segundo método.

En el segundo capítulo plantearemos el problema del agente viajero de dos formas, como un modelo de redes y como un programa entero mixto. A partir del modelo de redes, proporcionaremos una transformación del PM al PAV. Del planteamiento entero mixto del PAV, veremos que podemos resolver una relajación del problema en tiempo eficiente. A dicha relajación se le conoce como el problema de asignación y al algoritmo que lo resuelve se le llama húngaro. A partir de lo anterior, proporcionaremos un algoritmo de ramificación y acotamiento que resuelve el PAV apoyándonos además de algunos algoritmos de redes.

En el tercer capítulo examinaremos los errores comunes de la secuenciación de Illumina, razón por la cual es necesario definir una medida de calidad para la lectura de las bases nitrogenadas. Además examinaremos un estándar para almacenar los datos experimentales en una computadora, el cual es el formato `.fastq`. Posteriormente definiremos formalmente a las cadenas de caracteres, a sus prefijos, sufijos y traslapes entre cadenas. A partir de ello, propondremos un modelo del PM el cual nos proporcio-

nará el orden de las secuencias obtenidas. Este primer modelo no considerará errores de secuenciación, por lo que definiremos un segundo modelo el cual sí las tome en cuenta a partir de una penalización, la cual depende de la similitud entre los traslapes.

En el cuarto capítulo se presentará una introducción a la teoría de complejidad; daremos la definición de algoritmo a partir del modelo computacional de Turing para definir a la clase de problemas de decisión P y NP. De ahí definiremos a los problemas NP-Completos, NP de optimización y NP-Duros, para finalmente justificar que el problema del agente viajero es difícil de resolver, es decir, demostraremos que es NP–Duro. Para la justificación de la proposición anterior, utilizaremos reducciones polinomiales, reducciones de Turing y el Teorema de Cook–Levin. Debido a lo previo expuesto, introduciremos los métodos heurísticos y describiremos cualidades que buscamos en dichos. De aquí, hablaremos un poco de los algoritmos de tipo genéticos y describiremos uno para resolver instancias del PAV.

En el quinto capítulo trabajaremos con un ADN de referencia del fago λ. A partir de él, generaremos datos sin errores por medio de una computadora, aplicaremos el modelo propuesto en el capítulo 3 y lo secuenciaremos con los algoritmos presentados en los capítulos 2 y 4 para comparar los resultados con la secuencia original.

Adicionalmente se incluyen dos apéndices al final del trabajo. El primer apéndice incluye algunos elementos esenciales de teoría de redes y programación matemática que son indispensables para algunos resultados de los capítulos 2 y 4. En el segundo apéndice se incluye el pseudocódigo de las rutinas vistas en los capítulos 2, 3 y 4, así como dos diagramas del método de ramificación y acotamiento y del algoritmo genético.

Se pretendió que el texto fuera autocontenido. A continuación se muestra una lista y un diagrama de subsecciones para lectura rápida.

1.2.2 Ácidos desoxirribonucleicos.

1.3.2 Secuenciación de Illumina.

2.1.2 El problema del mensajero.

2.2.2 El algoritmo húngaro.

2.3.3 Ramificación y acotamiento para el PAV.

3.1.2 Formato FASTQ.

3.2.2 Red de traslapes.

3.3.2 Una métrica de errores.

3.3.3 Red de traslapes aproximados.

4.1.2 Problemas P y NP

4.1.3 Problemas NP-Completos y NP-Duros.

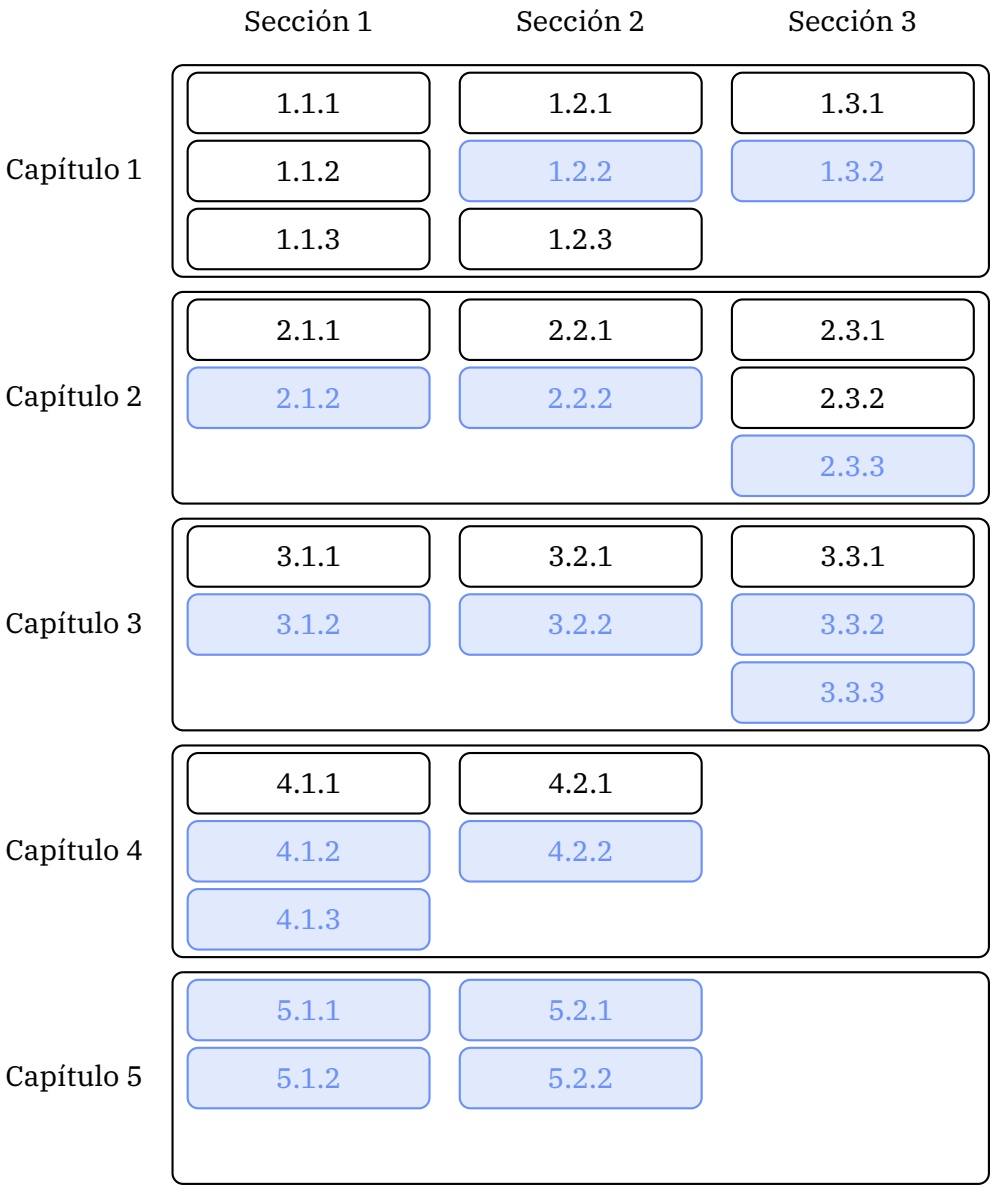
4.2.2 Algoritmos genéticos.

5.1.1 Creación del archivo FASTQ.

5.1.2 Limpieza de datos y matriz de costos.

5.2.1 Algoritmo de ramificación y acotamiento.

5.2.2 Algoritmo genético.


Sección 1
Sección 2
Sección 3
Capítulo 1
1.1.1
1.1.2
1.1.3
1.2.1
1.2.2
1.2.3
1.3.1
1.3.2
Capítulo 2
2.1.1
2.1.2
2.2.1
2.2.2
2.3.1
2.3.2
2.3.3
Capítulo 3
3.1.1
3.1.2
3.2.1
3.2.2
3.3.1
3.3.2
3.3.3
Capítulo 4
4.1.1
4.1.2
4.1.3
4.2.1
4.2.2
Capítulo 5
5.1.1
5.1.2
5.2.1
5.2.2

Capítulo 1

Secuenciación del ADN

En el libro *Biología*, Curtis, Barnes, Schnek y col. [2, pág. 172] afirman que las moléculas de ácido desoxirribonucleico tienen forma de hélice y portan parte de las instrucciones genéticas de cada ser vivo. Estas moléculas se encuentran distribuidas al interior del núcleo en el caso de las células eucariotas, y estas en una etapa del ciclo celular son envueltas en 23 pares de cromosomas en el caso del ser humano. A todo el conjunto de cromosomas se le conoce como **genoma** [2, pág. 215].

Este primer capítulo está dividido en tres secciones. En la primera sección daremos una breve introducción a la estructura de las moléculas orgánicas, de las cuales están conformados los seres vivos. En la segunda sección veremos a los ácidos nucleicos, su replicación y el procedimiento en el cual se interpreta al ADN para formar las moléculas que conforman a los seres vivos. En la tercera sección conoceremos dos importantes métodos para obtener la secuencia de fragmentos de ADN, los cuales son necesarios para obtener la secuencia del genoma.

1.1. Principios de biología molecular

Para comprender el tema principal del trabajo, es necesario que examinemos algunos conocimientos básicos de la biología, específicamente, la biología molecular. En su libro *Biología celular*, Avers [5] explica que las moléculas de ADN se encuentran albergadas de manera individual en el núcleo de una **célula** en el caso de las eucariotas, la cual es la unidad fundamental de los seres vivos. Las células suelen estar en constante cambio por medio de **reacciones químicas**. Recordemos que una reacción química es un procesos en cual los compuestos químicos (sustratos) cambian su estructura molecular para obtener nuevas moléculas (productos).

1.1.1. Enlaces químicos y moléculas orgánicas

Un **compuesto químico** es la unión de dos o más átomos, los cuales tienen la posibilidad de estar separados o unidos. Los átomos suelen mantenerse unidos por medio de **enlaces químicos**. Los enlaces químicos se clasifican en dos categorías: los fuertes que requieren mayor energía para su ruptura y los débiles que necesitan poca energía para ser rotos. Sólo necesitamos conocer los covalentes y los puentes de hidrógeno que

son enlaces de tipo fuertes y débiles respectivamente.

Los **enlaces covalentes** se forman cuando dos átomos comparten uno o más electrones. Esto se debe a que la estructura de dos átomos al unirse se vuelve energéticamente estable. Una molécula compuesta por enlaces covalentes tiene carga eléctrica neutra. Es posible tener hasta un enlace covalente triple que una a dos átomos. En la figura 1.1 se muestra la unión de un átomo de hidrógeno con un átomo de carbono por medio de un enlace covalente simple y la unión de un carbono con un nitrógeno por medio de uno triple. Los enlaces covalentes de dividen en dos grupos: los polares y no polares.

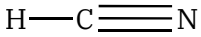


Figura 1.1: Representación de enlaces covalentes.

Los enlaces covalentes no polares se dan cuando los electrones de las dos moléculas están distribuidos uniformemente, un ejemplo es el hidrógeno molecular H_2 (H–H), el enlace de dicha molécula es apolar, ya que los electrones compartidos están distribuidos uniformemente. Lo contrario sucede con enlaces covalentes polares, los electrones están distribuidos de una manera no uniforme, lo que lleva a que dado un enlace apolar entre dos átomos, la carga eléctrica esté más cerca de un núcleo que de otro. Aunque los átomos estén equilibrados, intramolecularmente un átomo resulta tener mayor electronegatividad que otro.

Los **puentes de hidrógeno** son enlaces intermoleculares formados por un átomo de hidrógeno y un átomo con un enlace polar. Este enlace se forma por el núcleo del hidrógeno y la electronegatividad del átomo adherido al enlace polar. Usualmente el átomo suele ser de oxígeno, de nitrógeno o de flúor. En la figura 1.2 se muestra un ejemplo de un puente de hidrógeno entre dos moléculas de agua representado con la línea punteada.

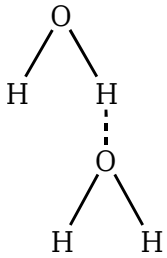


Figura 1.2: Puente de hidrógeno entre dos moléculas de agua.

Se denomina como **moléculas o compuestos orgánicos** a las estructuras moleculares cuya composición es mayormente de carbono. La cantidad de moléculas orgánicas es enorme, ya que el carbono tiene la habilidad de formar enlaces covalentes simples, dobles o triples con átomos de carbono, oxígeno, nitrógeno, hidrógeno, fósforo, azufre, entre otros. De esta forma, se pueden crear cadenas o estructuras cíclicas. Lo anterior se debe a que un átomo de carbono tiene 4 electrones de valencia y para poder estabili-

zarse se necesitan 4 electrones adicionales.

Las cuatro clases principales de compuestos orgánicos en la biología molecular son los carbohidratos, los lípidos, las proteínas y los ácidos nucleicos. Estas cuatro clases de compuestos orgánicos contribuyen a las estructuras, funciones y regulación de las células. Usualmente, las moléculas orgánicas son polímeros, los cuales son estructuras orgánicas de gran masa. Estos están compuestos por estructuras más pequeñas llamadas monómeros.

1.1.2. Proteínas

Las proteínas son moléculas orgánicas que ayudan a dar estructura y motricidad a los seres vivos, también son las encargadas de múltiples procesos de regulación, transporte y señalización a nivel celular. Antes de hablar de la estructura de las proteínas, debemos de conocer los aminoácidos y los polipéptidos. Los **aminoácidos** son monómeros formados por un grupo amino (NH_3^+), un grupo carboxilo (COO^-) y otra estructura molecular denotada por R (veamos la figura 1.3). Los aminoácidos se clasifican por la molécula R que se encuentre en su estructura, llamada **molécula lateral**. Se conocen muchos tipos de aminoácidos, pero 20 son necesarios para la generación de proteínas [5, pág. 42]. De dichos 20 aminoácidos, 10 pueden ser sintetizados por la célula (no esenciales) y 10 se obtienen por un medio externo (esenciales). Como ejemplos, un aminoácido esencial es la leucina (Leu) y un no esencial es la serina (Ser).

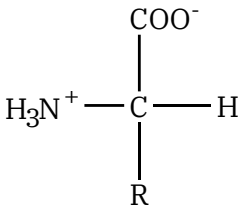


Figura 1.3: Estructura básica de los aminoácidos.

Los aminoácidos se pueden unir a través de enlaces covalentes llamados **enlaces peptídicos**, los cuales son el resultado de unir un grupo amino y un grupo carboxilo. Un ejemplo simple de un enlace peptídico es cuando a partir de unir dos aminoácidos se obtiene un dipéptido y una molécula de agua. Un dipéptido es ilustrado en la la figura 1.4, donde el enlace peptídico se muestra en color azul. A una molécula constituida por varios aminoácidos por medio de enlaces peptídicos, se le conoce como **péptido**. Usualmente los péptidos formados por no más de 10 aminoácidos, se les conoce como oligopéptidos, en otro caso, a dicho péptido se le dice **polipéptido**.

Estructuralmente, las proteínas son un polipéptido o varias cadenas polipeptídicas unidas por diversas fuerzas. Las proteínas llegan a tener cuatro descripciones estructurales distintas. La primera es conocida como **estructura primaria**, la cual es la secuencia (o secuencias) única de aminoácidos para cada proteína. Dicha secuencia suele ser representada con una cadena de círculos, que representan aminoácidos, conectadas por líneas, que representan a los enlaces peptídicos. En la figura 1.5a) se da una representación visual de la estructura primaria. Cabe recalcar que los aminoá-

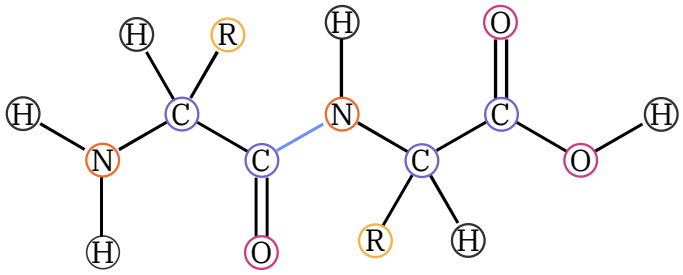
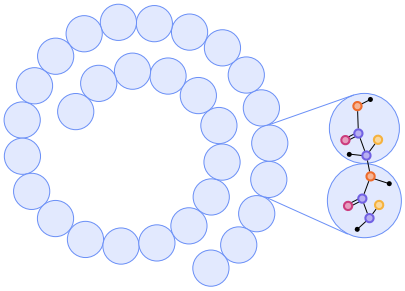


Figura 1.4: Estructura de un dipéptido.

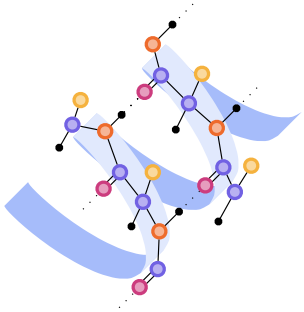
cidos que la proteína contenga y el orden de dichos aminoácidos, es esencial para el funcionamiento de cada proteína, ya que el mínimo cambio podría cambiar totalmente su función o sus funciones.

La estructura primaria no es suficiente para poder estudiar a las proteínas, pues su configuración espacial influye en su funcionamiento. La **estructura secundaria** nos permite ver cómo interactúan los aminos y carboxilos de diferentes aminoácidos en una proteína. Estos suelen conectarse por medio de puentes de hidrógeno por medio de un hidrógeno algún amino y el carbono de algún carboxilo. Como consecuencia de esto, algunos segmentos de las proteínas suelen doblarse usualmente en forma de hélice (α) o en forma de lámina (β). En la figura 1.5b) se muestra un ejemplo de hélice–α.

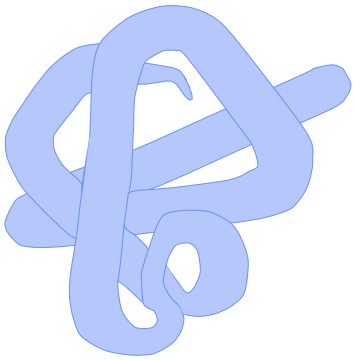
La **estructura terciaria** nos permite ver cómo interactúan las moléculas laterales de una proteína entre sí. Las moléculas laterales suelen interactuar por medio de puentes de hidrógeno u otras fuerzas, como los puentes disulfuro o interacciones iónicas. Las estructuras terciarias usuales son glóbulos y filamentos o fibras. En la figura 1.5c) se muestra un péptido plegado que conforma a la hemoglobina. Finalmente, en caso de tener una proteína conformada por varias cadenas polipeptídicas, necesitamos de una **estructura cuaternaria**, la cual especifica por colores a las subunidades de la proteína. En la figura 1.5d) aparecen todos los péptidos que constituyen a la hemoglobina.



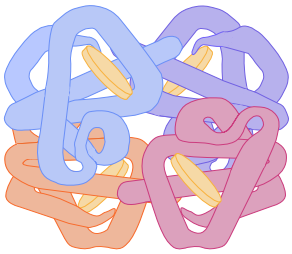
(a) Estructura primaria



(b) Estructura secundaria



(c) Estructura terciaria



(d) Estructura cuaternaria

Figura 1.5: Diferentes estructuras de la hemoglobina.

1.1.3. Enzimas

Las interacciones dentro de la célula quedan definidas por una serie de reacciones químicas. A esta química celular, se le denomina **metabolismo**. Para comprender su funcionamiento, es necesario conocer tres aspectos: el origen de la energía necesaria para la realización de las reacciones químicas; los mecanismos encargados de la transferencia de la energía y las moléculas que permiten o facilitan la ejecución de las reacciones. A las últimas moléculas mencionadas, se les conoce como **catalizadores**. Existen cierta clase de catalizadores biológicos que son de mayor índole para este trabajo, los cuales son llamados **enzimas**.

Las enzimas son proteínas que influyen en la velocidad de las reacciones químicas. Toda reacción química necesita energía para poder ser llevada a cabo. A la energía mínima necesaria para efectuar una reacción, se le conoce como **energía de activación**. La enzima permite que la energía de activación sea reducida. Visto desde una perspectiva espacial, las moléculas dispersas en un medio, tienen cierta probabilidad de reaccionar unas con otras. Como las enzimas reducen la energía de activación, aumenta la probabilidad de que dichas reacciones sucedan en el medio. Además, las enzimas no benefician a que ocurran todas las posibles reacciones de un medio, pues éstas suelen

ser altamente específicas. Mientras una enzima es más específica, será más rápida la reacción ya que la capacidad del sustrato para discriminar moléculas ajenas será mayor.

Algo importante es destacar que tras la reacción enzimática, la enzima o las enzimas involucradas se mantienen intactas. Una analogía muy usada para describir el mecanismo de las reacciones enzimáticas, es la famosa relación de *cerradura y llave*, esto se debe a que muchas enzimas son tan buenas al diferenciar a un sustrato de otras moléculas en el medio. Esto se debe a que los sustratos tienen un sitio molecular específico al cual se adhiere la enzima. La enzima también cuenta con una región llamada **sitio activo**, que es el lugar al que se une la enzima con el sustrato. En síntesis, la reacción enzimática consiste de dos pasos, la enzima y el sustrato se encuentran en el medio y se incorporan para formar un complejo transitorio *enzima-sustrato*, posteriormente se produce la reacción química para obtener los productos y la enzima se separa de dichos productos. En la figura 1.6 podemos apreciar dichos pasos.

Un medio de control en los sistemas biológicos, es reduciendo o neutralizando las reacciones enzimáticas. Dicha reducción se puede lograr al inhibir las enzimas. Hay dos tipos de inhibiciones, las irreversibles y las reversibles. Por una parte, la **inhibición irreversible** puede darse al cambiar la estructura de una enzima al adherir un inhibidor a la molécula. El proceso de disociación entre el inhibidor y la enzima suele ser muy tardado. Por otra parte, en la **inhibición reversible** la enzima no pierde sus propiedades funcionales y la inhibición puede ser anulada veloz.

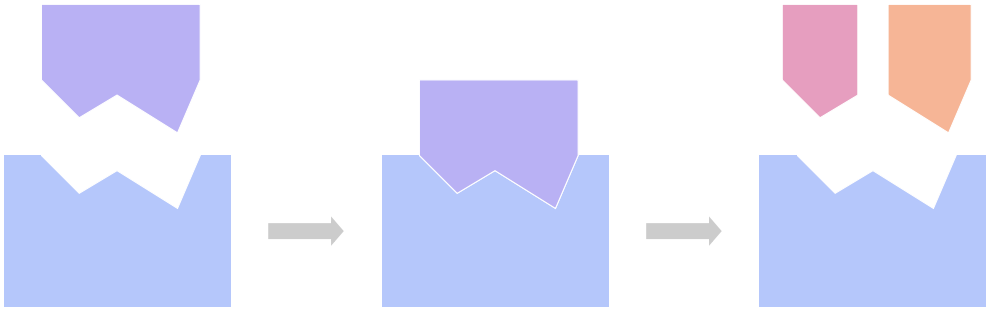


Figura 1.6: Pasos de una reacción enzimática con la relación cerradura–llave.

1.2. Los ácidos nucléicos y el dogma central

Curtis, Barnes, Schnek y col. [2] y Avers [5] mencionan que una de las moléculas más importantes en la biología son los ácidos nucleicos, ya que dentro de ellas se encuentran los genes. Un gen está definido como una secuencia de ácidos nucleicos que portan las instrucciones para sintetizar una proteína. Hay dos tipos de ácidos nucleicos, los ácidos desoxirribonucleicos (ADN) y los ácidos ribonucleicos (ARN). A lo largo de esta sección veremos su composición, sus principales características y el dogma central de la biología molecular.

1.2.1. Nucleótidos

Avers [5] afirma que así como las proteínas están formadas en cadena por aminoácidos, las secuencias de ADN están formadas por moléculas llamadas nucleótidos. Cada **mononucleótido** está formado por un grupo fostato, una azúcar pentosa y una base nitrogenada. El nombre de las bases nitrogenadas es debido a que son compuestos orgánicos formados por al menos dos átomos de nitrógeno. Hay muchos tipos de bases nitrogenadas, pero nos interesan sólo las siguientes: *adenina, timina, uracilo, guanina y citosina*. Las letras características del ADN son dadas por la inicial de cada base nitrogenada. Cabe recalcar que la timina se encuentra en el ADN y el uracilo en el ARN. La adenina y guanina se llaman **bases púricas** y se caracterizan por su estructura de dos anillos. La timina, el uracilo y la citosina son **bases pirimidínicas** y se caracterizan por tener un anillo en su estructura.

Los **polinucleótidos** son formados por múltiples mononucleótidos unidos por un enlace covalente llamado **puente fosfodiéster**. Los puentes fosfodiéster unen a dos azúcares pentosa en el tercer (3') y quinto (5') carbono por medio de un grupo fosfato. Gracias a las propiedades de los nucleótidos, hay 4^n posibles polinucleótidos de longitud n, ya que tenemos 4 bases nitrogenadas para el ADN (A,C,G,T) y para el ARN (A,C,G,U). La adenina y timina, así como la guanina y citosina, tienen la posibilidad de formar puentes de hidrógeno mediante los cuales se mantienen unidas dichas moléculas, por eso se llaman bases complementarias. En la figura 1.7 se presenta a los nucleótidos del ADN, así como la dirección de lectura y las tres componentes principales de un mononucleótido. Los vértices de dos líneas sin especificar al elemento, representan un átomo de carbono.

1.2.2. Ácidos desoxirribonucleicos

Como ya expusimos previamente, Curtis, Barnes, Schnek y col. [2] comentan que las instrucciones genéticas de cada individuo están descritas en las moléculas de ADN. Dentro de cada célula, podemos encontrar moléculas de ADN. En el caso de la célula eucariota, varias cadenas de ADN se encuentran en su núcleo. Una de las principales características de esta molécula, es que tiene una estructura de doble hélice (dextrógira), gracias a la complementariedad de las bases. Esto permite que se mantenga la continuidad genética al momento de replicar las moléculas.

El ADN se replica de forma **semiconservativa**. Dicha proceso utiliza las dos hebras de una secuencia como molde y como resultado se obtienen dos copias. El procedimiento sucede durante el crecimiento celular y es parte de su reproducción. Éste empieza con la separación de los puentes de hidrógeno en un extremo, los cuales mantienen unidas a las dos cadenas. La separación es llevada a cabo en varios sitios específicos conocidos como *origen de la replicación*. El proceso requiere de diferentes proteínas y enzimas, que ayudarán a la separación y conexión de las hebras, colocación de los mononucleótidos, unión de los diversos fragmentos (de hebras) que se repliquen, entre otras funciones.

Dada una secuencia a replicar, es necesario tener una molécula conocida como *primer* o *cebador*, la cual es una pequeña secuencia de ARN que se acoplará al sitio

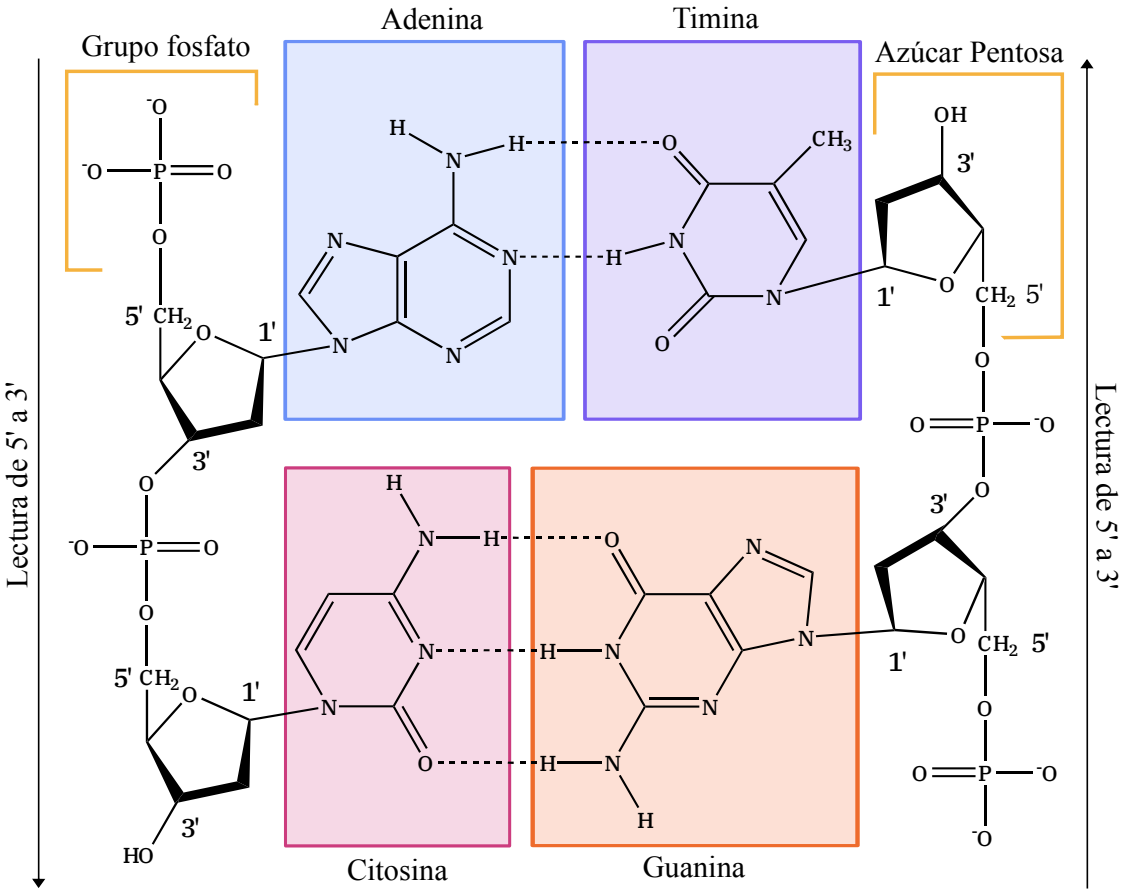


Figura 1.7: Molécula de ADN.

de iniciación de la replicación. Tras esto, la ADN polimerasa se adhiere a la cadena y empieza a sintetizar a las nuevas moléculas. Lo anterior se realiza a la par de que las helicasas desenrollan el ADN y rompen los puentes de hidrógeno y las proteínas de unión mantienen ambas cadenas separadas. La síntesis se realiza en dirección 5’ a 3’ de forma continua y de 3’ a 5’ de forma discontinua. Durante el proceso de replicación, los primers son sustituidos por mononucleótidos de ADN y los fragmentos (de Okazaki) sintetizados en dirección 3’ a 5’ de manera discontinua se separan del mecanismo de replicación con sus extremos protegidos con proteínas. Posteriormente, los fragmentos son presentados a una hebra continua y acoplados por las ADN ligasas. Al terminar el copiado, la molécula es liberada y otra hebra es replicada sucesivamente hasta duplicar a todo el genoma.

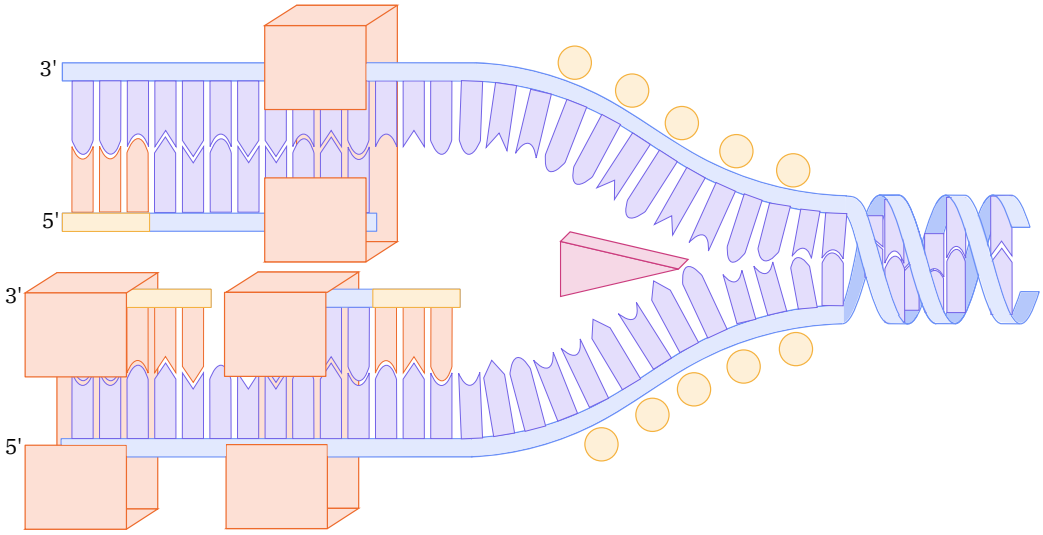


Figura 1.8: Replicación de ADN.

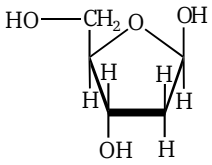
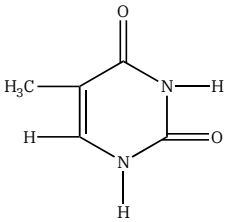
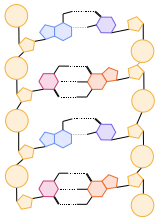
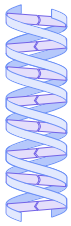
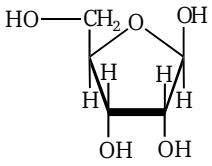
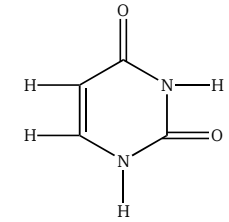
Al momento de replicar una secuencia, la polimerasa suele tener errores en la colocación de los mononucleóticos. Este error se da cada cien mil pares de bases. Hay mecanismos que se encargan de corregir estos desaciertos y logran que no haya errores en cada diez millones de pares de bases o incluso en diez mil millones. Para ponerlo a comparación, consideremos que el genoma humano tiene 3 mil millones de pares de bases. Aún con las mejores tasas de error, siempre es posible tener errores durante la replicación. Dichos errores se denominan **mutaciones**.

Hay cuatro propiedades importantes del ADN: 1) se pueden generar replicas exactas de la molécula; 2) sufre mutaciones y las transmite a futuras generaciones; 3) porta la información genética que especifica las características de las células y los organismos; 4) y transfiere la información genética a moléculas que permitan la síntesis de las proteínas. La síntesis de las proteínas es realizada afuera del núcleo en las células eucariotas, por lo que se necesita de una molécula que transmita esta información.

1.2.3. Ácidos ribonucleicos y el dogma central de la biología molecular

El ARN es otro ácido nucleico esencial para los procesos biológicos. A diferencia del ADN, el ARN consta de una sola cadena en su estructura. Otra diferencia es el azúcar pentosa, ya que en el ADN tenemos una azúcar desoxirribosa, mientras que en el ARN es la ribosa. En la tabla 1.1, se encuentran ilustradas las diferencias. Tres principales variantes del ARN implicadas en la síntesis de proteínas, son el ARN mensajero (ARNm), el ARN traductor (ARNt) y el ARN ribosómico (ARNr).

Tabla 1.1: Diferencias entre el ADN y el ARN.

	Azúcar	Base pirimídica	Conformación lineal	Plegamiento espacial
ADN				
ARN				

La labor del ARNm es portar la información de un gen desde el núcleo (en células eucariotas) al citosol, lugar donde se sintetizan las proteínas por medio de moléculas llamadas *ribosomas*. El proceso descrito a continuación, es llamado transcripción. El ARNm es elaborado en el núcleo por una ARN polimerasa durante la activación de un gen en algún cromosoma. El inicio del gen es llamado *promotor*, el cual es una secuencia de nucleótidos. La ARN polimerasa se adhiere al inicio del gen en una de las dos cadenas, separa a las dos hebras y a partir de bases dispersas en el medio, comienza a elaborar una cadena de ARNm en dirección 5' a 3' mientras que la ARN polimerasa se mueve en dirección 3' a 5' sobre la hebra de ADN. Para finalizar con la elaboración del ARNm, el ARN libera a la hebra de ADN y la nueva molécula de ARNm. Esto sucede gracias a una secuencia de *terminación* y por medio de una enzima externa o al haber un plegamiento en el ARNm.

Antes de salir del núcleo, el ARNm necesita pasar por una etapa de procesamiento, donde modificará a la molécula. Al inicio de la molécula se adherirá una guanina modificada conocida como *5' casquete*, mientras que al final se añadirá una molécula llamada *cola poli-A* conformada de múltiples adeninas. Las moléculas anteriores previenen que el ARNm sufra daños en su trayecto a los ribosomas. Adicionalmente a estos cambios, hay fragmentos de la secuencia (intrones) de ARNm que son retirados, mientras que el resto de fragmentos (extrones) permanecen en la molécula. Es posible que el pro-

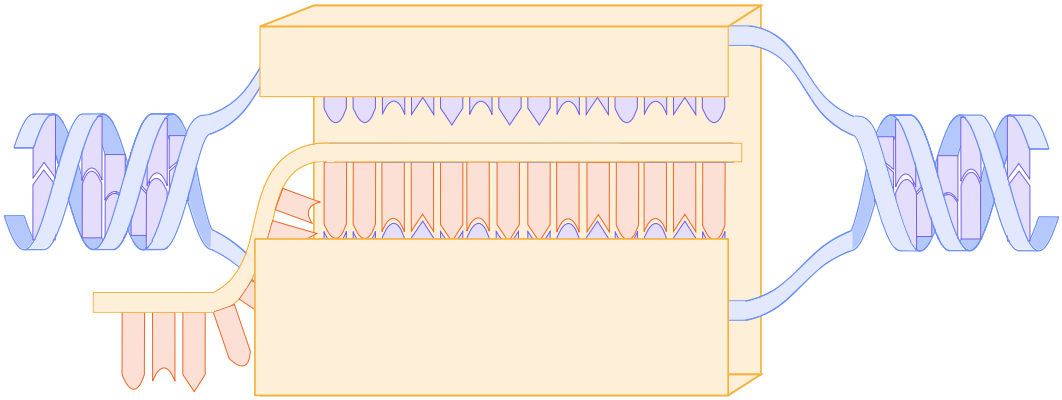


Figura 1.9: Transcripción de un gen.

cesamiento suceda durante la transcripción. Cabe mencionar que al igual que en la transcripción, la traducción también es realizada con la ayuda de enzimas y proteínas.

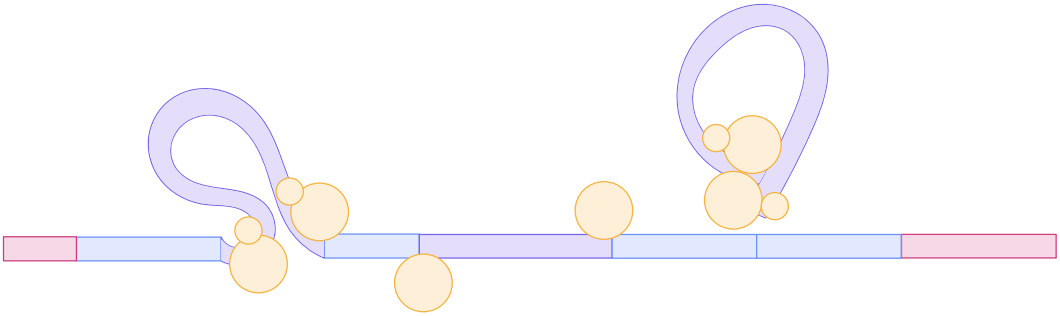


Figura 1.10: Maduración del ARNm.

La última etapa para la síntesis de proteínas es un proceso conocido como *traducción*, aquí el ARNm es utilizado por un ribosoma para formar un polipéptido. El ARNr y el ARNt juegan papeles esenciales en esta etapa. Los ribosomas están conformados por subunidades constituidas por proteínas y ARNr. Cada uno tiene tres sitios donde pasará el ARNm, el sitio A, el sitio P y el sitio E. El ARNt es una estructura que transporta un aminoácido a los ribosomas. Recordemos que 20 aminoácidos son esenciales para la formación de polipéptidos, y 4 bases nitrogenadas no son suficientes para indicar el orden de aminoácidos. Un **codón** es una secuencia de tres bases nitrogenadas, la cual tiene asociada a un aminoácido. Hay 64 posibles secuencias de 3 aminoácidos, las

cuales son suficientes para especificar a los 20 aminoácidos esenciales. El ARNt tendrá un codón complementario para cada codón dentro del ARNm.

En la inicialización de la traducción, el ARNm se acopla al sitio P del ribosoma. Específicamente, un codón AUG, conocido como el codón de iniciación, se adhiere al ribosoma. Un ARNt con codón complementario UAC con metionina (un aminoácido) se acopla al ARNm y a partir de aquí, empieza la construcción del péptido. Al siguiente codón de la secuencia se acopla otra molécula de ARNt, la cual es colocada en el sitio A. En el momento que el enlace peptídico sea formado, las moléculas en el sitio T son desplazadas al sitio E y las moléculas en el sitio A son desplazadas al sitio T. El ARNt en el sitio E se desacopla del ribosoma y de su aminoácido. Este procedimiento continúa hasta que al ribosoma llegue una secuencia de terminación por parte del ARNm. Hay tres secuencias de terminación: UAA, UAG y UGA. A dichas secuencias se acoplan un ARNt de terminación, el cual no contiene aminoácido alguno. Cuando algún ARNt con secuencia complementaria AUU, AUC o ACU se acople al ribosoma, la secuencia de ARNm y la cadena polipeptídica serán liberados y las subunidades ribosomales se disocian.

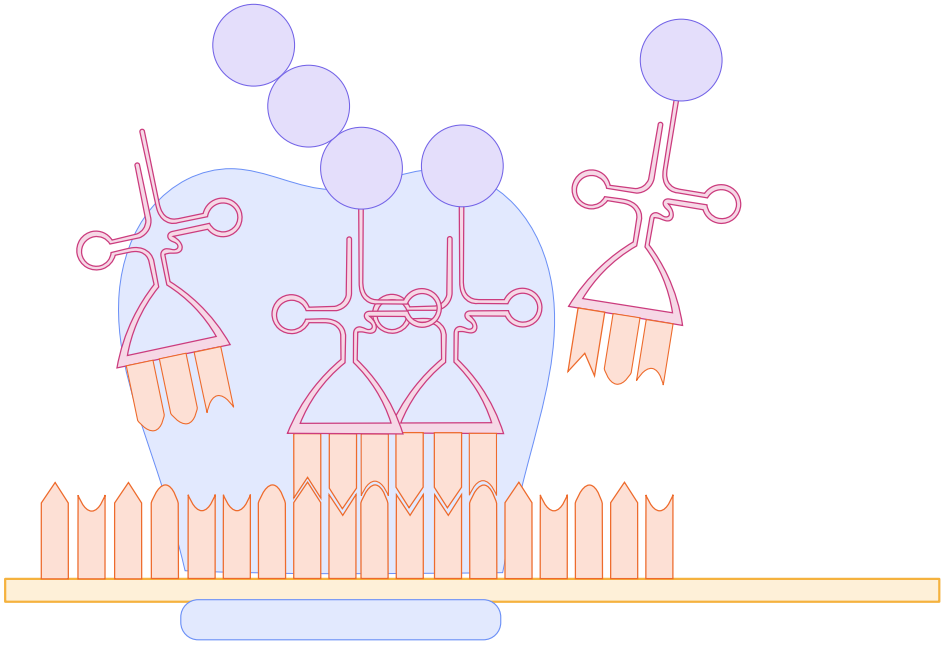


Figura 1.11: Traducción del ARNm a proteína.

Recapitulemos lo visto hasta el momento sobre el flujo de información genética: 1) las secuencias de ADN son *replicables*; 2) el ADN es apto para *transcribirse* a ARN; 3) el ARN es *traducido* en proteínas. A este flujo unidireccional se le conoce como **el dogma central de la biología molecular**. El dogma fue propuesto y bautizado por Francis Crick, un codescubridor de la molécula del ADN. Aunque el proceso sea conocido de dicha forma, muchos científicos prefieren el nombre de *hipótesis central*, ya que la palabra dogma se refiere a una afirmación que no debe ponerse en duda. No todos los organismos compuestos por ácidos nucléicos siguen este dogma, algunas excepciones son los ribosomas y los virus. Las excepciones involucran procesos de transcripción

inversa (De ARN a ADN) y autoreplicación de ARN.

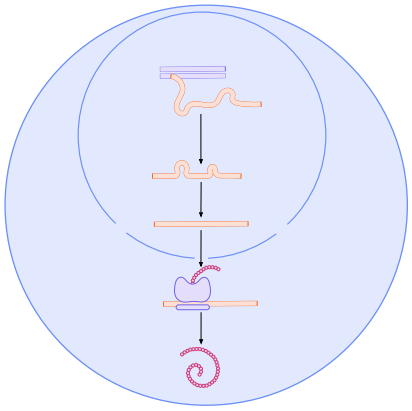


Figura 1.12: El dogma central de la biología molecular.

1.3. Secuenciación de lecturas

Conocer las secuencias de aminoácidos es importante en muchas ramas de la biología y medicina. De acuerdo con Heather y Chain [3], durante las últimas décadas, se han desarrollado varios métodos para conocer el orden de los nucleótidos en una secuencia de ADN o ARN. Hasta el momento, se han desarrollado tres generaciones diferentes de tecnologías para llevar a cabo el procedimiento mencionado. Una de las mayores dificultades al momento de secuenciar, es que a mayor longitud de la secuencia, más complicaciones hay en el proceso. El procedimiento para una cadena larga de ADN es realizar varios cortes y posteriormente encontrar la disposición de las bases nitrogenadas.

Veamos un procedimiento fundamental mencionado por Curtis, Barnes, Schnek y col. [2]. Cuando una molécula de ADN es sometida a una temperatura de 95º, se da la ruptura de los puentes de hidrógeno y esto ocasiona que las hebras se separen. Si posteriormente se reduce la temperatura a 60º y se agrega cebadores, monodesoxiribonucleótidos y polimerasas al medio, un proceso de replicación sería llevado a cabo. Los cebadores se incorporan a cada hebra, la polimerasa se une a los cebadores y comienza la formación de las cadenas complementarias. A este procedimiento se le conoce como la **reacción en cadena de la polimerasa (PCR)** y es esencial para la secuenciación de ADN. A una repetición del procedimiento anterior se le llama **ciclo de amplificación de la PCR**. Si este proceso es repetido sucesivamente, tendríamos como resultado varias copias recortadas del ADN con el que contábamos en un inicio. El recorte es debido a que los primers se adhieren a sitios específicos al interior de la molécula. Las ventajas de este procedimiento es que no es costoso y podemos realizar varias copias de ADN en poco tiempo.

Como ya mencionamos, en caso de contar con cadenas largas, es necesario recortarlas en trozos más pequeños para que sea posible su secuenciación, como mencionan Alberts, Johnson, Lewis y col. [6]. Esto se puede realizar por medio de una separación

física o por medio de la ADNasa o la ARNasa, que son enzimas capaces de romper el enlace que une al carbono 3' de una azúcar pentosa y a un grupo fosfato. Estos cortes físicos son elaborados de manera aleatoria pero es posible que se efectúen de tal manera que las hebras resultantes aproximen cierta longitud. La ADNasa buscará un sitio específico (una secuencia corta de mononucleótidos) donde efectuará el corte. A estas hebras de longitud recortada se les conoce como **lecturas** y juegan un papel fundamental para conocer la secuencia original. Para finalizar el capítulo, veremos dos tipos de secuenciación de lecturas.

1.3.1. Secuenciación Sanger

Heather y Chain [3] comentan que uno de los primeros métodos de secuenciación es el **método Sanger**. Recordemos que los mononucleótidos del ADN tienen un extremo 3' y un extremo 5'. Estos son llamados deoxinucleótidos (dNTP's), donde la N denota al nucleótido (A, C, T o G). Al hacer la replicación de una lectura, la elaboración es en dirección 5' a 3'. La idea de la secuenciación Sanger es agregar al medio un nuevo tipo de mononucleótido, llamado dideoxinucleotido (ddNTP), que no tiene un grupo OH en su carbono 3' y esto previene la adición de un nuevo dNTP. Si la polimerasa llega a acoplar un ddNTP durante la replicación de una hebra, el proceso se detendrá y se liberarán ambas hebras.

Para realizar la secuenciación Sanger, se necesita un contenedor, al que se le agrega polimerasas, primers, copias de la lectura a replicar, dNTP's y ddNTP's. Los ddNTP's se agregan en menor concentración con el fin de que la polimerasa se detenga en todos los sitios. Adicionalmente los ddNTP's contienen tintes fluorescentes para ser visualizados al ser expuestos a un láser. Una vez agregados los químicos al contenedor, las polimerasas inician el proceso de replicación forman varias subcopias de distintas longitudes de las lecturas. El contenido del frasco se puede incorporar a un sistema de electroforesis capilar. Recordemos que los fragmentos más pequeños, recorren el gel más rápido. Al aplicar electricidad a dicho sistema, un láser y un lector detectarán la luz reflejada por los tintes fluorescentes. Se utiliza un tinte distinto para cada base. En la figura 1.13 se muestra un ejemplo para la lectura `TAAGCATGCA`.

1.3.2. Secuenciación de Illumina

Durante muchos años se utilizaron los métodos de primera generación para obtener la secuencia de lecturas [3]. Uno de los principales objetivos de los científicos, era reducir los costos de las secuenciaciones. Un gran avance se dio con los secuenciadores de próxima generación o secuenciadores de segunda generación. Estos disminuyeron los costos considerablemente al poder secuenciar varias lecturas al mismo tiempo. El método de segunda generación que describiremos, es la **secuenciación por síntesis (SBS)**. Dicho método de secuenciación es llevado a cabo en una máquina. Existen muchos secuenciadores de este tipo, pero en este trabajo describiremos la tecnología de *Illumina*, en particular la del secuenciador **MiSeq**.

Suponiendo que ya contamos con las lecturas, a ambos extremos de las lecturas se agrega un adaptador inicial de ADN y un adaptador final. Ambos contienen un primer, un índice de reconocimiento y un sitio de adherencia. Los sitios de adherencia iniciales

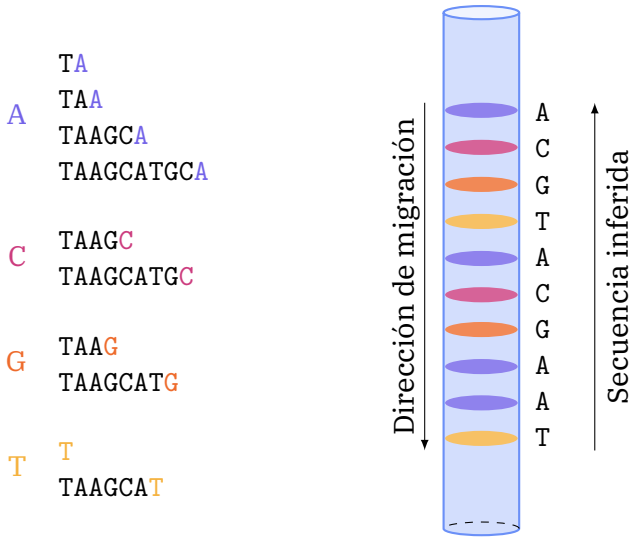


Figura 1.13: Secuenciación Sanger.

son iguales entre sí, así como los finales. Estos extremos son clave para un proceso llamado amplificación. Tras añadir adaptadores a las lecturas, éstas pasan por varias celdas de flujo, las cuales tienen copias de los adaptadores añadidos a los fragmentos. De esta forma, cada lectura se acopla sólo a un lugar específico de las celdas. En este sitio, comienza la **amplificación por puente**. La lectura es replicada, desnaturalizada y extraída del medio. La copia de dicha lectura se une al otro sitio de adherencia para realizar nuevamente un ciclo de secuenciación. Varios ciclos de amplificación son requeridos para obtener un **agrupamiento** de varias copias de la lectura, así como copias de su hebra complementaria. Ya con los ciclos concluidos, las copias con adaptador final acoplado a la celda son removidas y los adaptadores sin lecturas son protegidos para evitar el acoplamiento de otras moléculas. El procedimiento es presentado en la figura 1.14.

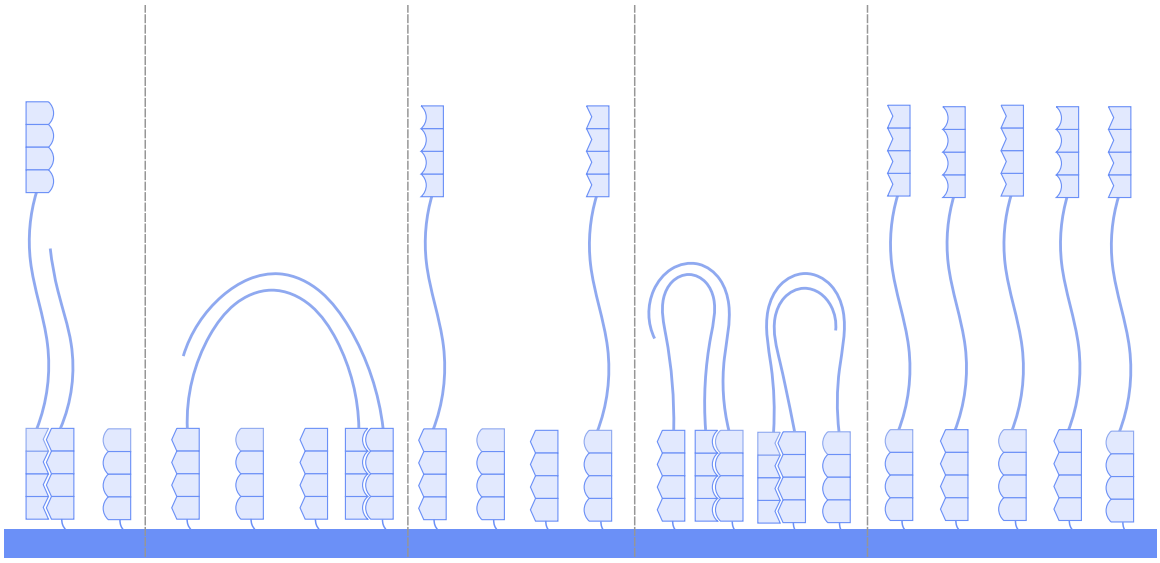


Figura 1.14: Amplificación de la lectura.

A partir de este punto, comienza el proceso de secuenciación. Varios primers iniciales son añadidos al medio. También se introducen nucleótidos con terminadores fluorescentes y ADN polimerasas para replicar las secuencias. Además de distinguir las bases nitrogenada por colores, los terminadores previenen que cada polimerasa sintetice la molécula rápidamente, esto con la finalidad de poder fotografiar toda la celda y así observar la base añadida en cada lectura. Para realizar la fotografía, se extraen las bases del medio. Tras hacer el retrato, el terminador es removido y se agregan las bases con terminadores fluorescentes. Decimos que ha pasado un **ciclo** en la SBS cada vez que se agrega un nucleotido, se realiza una fotografía y se retira el terminador fluorescente de la lectura. El proceso se repite hasta obtener la longitud deseada de la lectura, la cual quedará definida por el número de ciclos que se realicen. Esta reacción es ejemplificada en la figura 1.15.

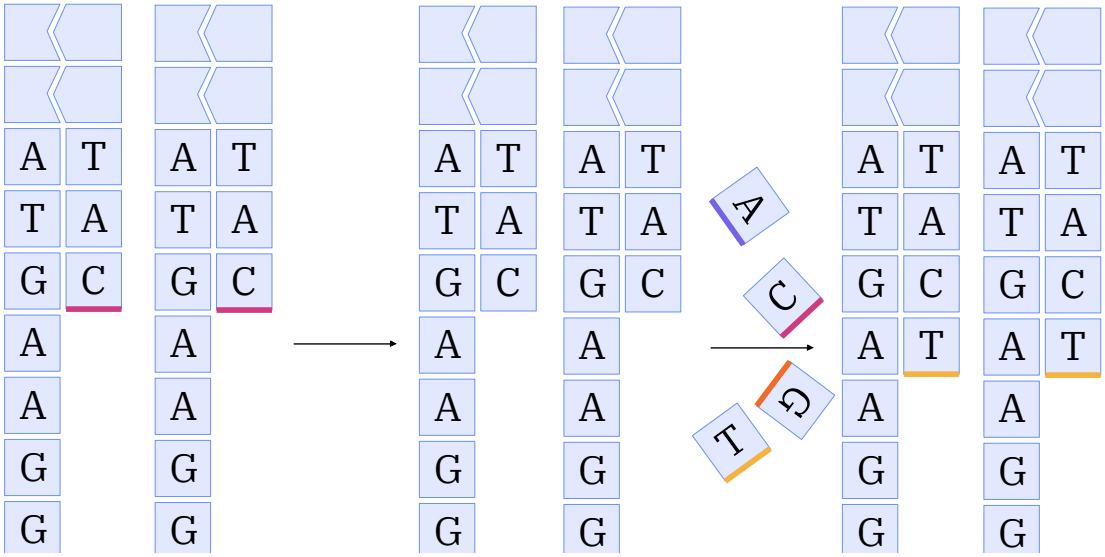


Figura 1.15: Un ciclo de la SBS.

El producto de la replicación es sustraído de la celda. Se añade un primer para el índice de la lectura, el cual es secuenciado de forma análoga. El producto de la replicación del índice sale de la celda. Recordemos que los adaptadores finales se encuentran protegidos, por lo que se retira la protección para que pueda unirse la secuencia acoplada al adaptador inicial. Nuevamente se replican las lecturas y ya con ambas copias, las copias con adaptadores iniciales acoplados son liberadas y los adaptadores son protegidos. Se repite el procedimiento para secuenciar la lectura complementaria y el índice. Al final, las lecturas son almacenadas en un archivo FASTQ para posteriormente ser analizadas.

El último paso es conocer el orden de la secuencia original a partir de la información de las lecturas. En el tercer capítulo hablaremos a detalle sobre cómo ordenar los datos obtenidos. Dicha secuencia puede ser obtenida con una secuencia de referencia o por medio de una secuenciación por escopeta, que es para secuencias sin referencias [1, pág. 159]. Podemos transformar el problema del ordenamiento de lecturas en el problema del agente viajero, definido en el siguiente capítulo.

Capítulo 2

El Problema del Agente Viajero

Para este punto ya tenemos las bases biológicas suficientes para entender cómo es el procedimiento experimental para secuenciar una molécula de ADN, pero no tenemos las bases matemáticas para el procedimiento informático. Es necesario conocer el problema del agente viajero (PAV) para saber ordenar las lecturas de ADN y obtener la secuenciación del genoma [1]. En síntesis, el problema del agente viajero consiste en recorrer un listado de ciudades de la manera más eficiente posible pasando por ellas sólo una vez y regresando a la ciudad de origen [7].

Aunque a primera vista parezca algo simple, el problema es aplicable en áreas como las ciencias genómicas, la lógica, las telecomunicaciones, entre otras. Durante muchas décadas, se han realizado grandes estudios respecto a este problema para conseguir métodos efectivos que lo resuelvan. En este capítulo, plantearemos el problema del agente viajero e implementaremos un método para poder resolver instancias del problema.

2.1. Conceptos Preliminares

El objetivo de esta sección es dar una introducción de los conceptos matemáticos para el entendimiento del PAV. Podemos utilizar elementos de teoría de redes y programación entera para plantear el problema. También veremos una variante del problema que será de nuestro interés en los capítulos posteriores. Antes de dar el planteamiento general, veamos un ejemplo práctico para motivar el estudio sistemático de los temas mencionados.

Ejemplo 2.1. Los pilotos aviadores Aníbal y Agustín trabajan para una empresa correos y paquetería con cede en la Ciudad de México. Por cuestiones de logística, la empresa sólo cuenta con ellos para repartir paquetes a las siguientes ciudades: Ciudad de México, Guadalajara, Monterrey, Los Cabos, Tijuana y Hermosillo. Dicha entrega debe de realizarse en el menor tiempo, regresando a Ciudad de México y sin pasar dos veces por una misma ciudad. El empleador de ambos pilotos quiere que las entregas sean efectuadas en un sólo día, pero sus contratos indican que pueden pilotar un total de 6 horas por día. Veamos si esto es posible. A continuación, en la tabla 2.1 se muestra el tiempo estimado en minutos de cuánto tardaría de ir de una ciudad a otra [8].

Tabla 2.1: El tiempo estimado de viaje entre una ciudad y otra en minutos.

	Destino					
Origen	CDMX	Guadalajara	Monterrey	Los Cabos	Tijuana	Hermosillo
CDMX	-	64	83	107	188	155
Tijuana	64	-	80	86	159	120
Monterrey	83	80	-	111	152	135
Los Cabos	107	86	111	-	114	97
Tijuana	188	159	152	114	-	141
Hermosillo	155	120	135	97	141	-

Podemos definir una gráfica $G = (X, A)$ donde X es el conjunto de ciudades que el piloto debe visitar, es decir, $X := \{1, 2, 3, 4, 5, 6\}$, donde 1 representaría a Ciudad de México, 2 a Guadalajara, 3 a Monterrey, 4 a Los Cabos, 5 a Tijuana y 6 a Hermosillo. Notemos que el conjunto de aristas sería el de las posibles rutas de vuelo que Aníbal puede tomar y estaría dado por $A := \{(i, j) \in X \times X : i \neq j\}$ pues no nos interesa considerar arcos que conecten a una ciudad consigo misma. Observemos que en este caso no necesitamos de una gráfica dirigida, pues la distancia de un nodo x a un nodo y es la misma que del nodo y al nodo x y por ello no es necesario definir direcciones. La gráfica quedaría descrita visualmente como se muestra en la figura 2.1.

Observación 2.1. En el contexto del PAV, se le conoce como **tour** a un ciclo o circuito hamiltoniano y a los circuitos o ciclos no hamiltonianos se les conoce como **subtours**.

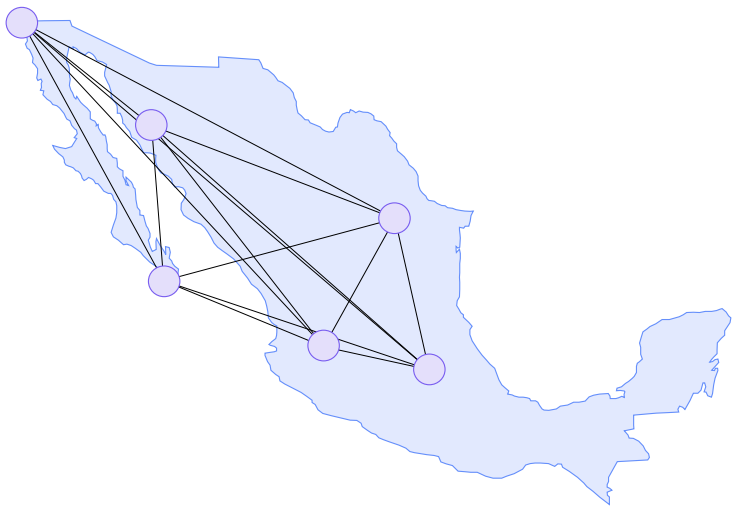


Figura 2.1: Gráfica de ciudades y rutas aéreas.

En este caso nuestro problema es encontrar el ciclo hamiltoniano de menor costo. Notemos que aún no incluimos los tiempos estimados de cada ruta de vuelo. Podemos definir una función $c : A \to \mathbb{R}$ que sería el tiempo promedio que le tomaría a Anibal ir de la cuidad i a la ciudad j. Dichos valores los podemos tomar de la tabla 2.1. Con los elementos anteriores, podemos definir una red $R = (X, A, c)$.

2.1.1. Planteamiento general

Consideremos un problema más general. Supongamos que tenemos un conjunto de n ciudades ($n \geq 3$), conectadas directamente una entre otra. Supongamos sin pérdida de generalidad que la ciudad de origen es la ciudad 1. El costo de ir de la ciudad i a la ciudad j es $c_{i,j} \geq 0$, con $i, j \in \{1, \ldots, n\}$ e $i \neq j$. Lo anterior puede ser planteado como un problema de programación lineal binaria. Las variables de decisión serían aquellas que nos indiquen si se viaja de la ciudad i a la ciudad j. Definimos a las variables de decisión $x_{i,j} \in \{0, 1\}$ como

$$x_{i,j} = \begin{cases} 1 & \text{si se viaja de la ciudad } i \text{ a la ciudad } j, \\ 0 & \text{e.o.c.} \end{cases}$$

con $i, j \in \{1, ..., n\}, i \neq j$. Para las restricciones, primero consideremos que al salir de una ciudad i, sólo podemos llegar a sola una ciudad j. Esto lo aseguramos con la restricción

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} x_{i,j} = 1, \quad i \in \{1, \ldots, n\}. \tag{2.1}$$

Además, si llegamos a una ciudad j es porque salimos de una única ciudad i, por lo que

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{i,j} = 1, \quad j \in \{1, \ldots, n\}. \tag{2.2}$$

Las restricciones anteriores no son suficientes para definir el problema, pues es posible tener subtours, los cuales son circuitos de cardinalidad menor a n. Consideremos la solución $x_{1,2}, x_{2,3}, x_{3,1}, x_{4,5}, x_{5,6}, x_{6,4} = 1$ y el resto de las $x_{i,j} = 0$ para el ejemplo 2.1, es factible considerando únicamente las dos restricciones anteriores. Esto indicaría que Anibal iría a las ciudades 1, 2 y 3 en un tour y posteriormente a las ciudades 4, 5 y 6.

Para evitar el problema anterior, es necesario introducir restricciones de eliminación de subtours. Dantzig, Fulkerson y Johnson [9] exhiben que una forma de expresar estas restricciones es evitando circuitos de cardinalidad menor a n que excluyan al nodo 1. Así,

$$\sum_{i \in Q} \sum_{\substack{j \in Q \\ j \neq i}} x_{i,j} \leq \text{card}(Q) - 1, \tag{2.3}$$

para todo conjunto $Q \in \mathscr{P}(\{2, \ldots, n\})$ tal que $\text{card}(Q) \geq 2$. Con lo anterior, todos los circuitos de cardinalidad menor o igual a $n-1$ no serán factibles, dejando sólo los circuitos hamiltonianos. El problema con estas restricciones, es que al trabajar con la potencia del conjunto $\{2, \ldots, n\}$, la cantidad de restricciones será de aproximadamente 2^n. Por lo tanto, el planteamiento anterior resulta impráctico.

Un planteamiento para el PAV con una menor cantidad de restricciones es propuesto por Miller, Tucker y Zemlin [10], donde ahora tenemos un problema lineal entero mixto. Para expresar las restricciones de eliminación de subtours, se definen las variables auxiliares $\alpha_i \in \mathbb{R}$, para $i \in \{2, \ldots, n\}$ y $n \geq 4$. Obtenemos las siguientes restricciones

$$\alpha_i - \alpha_j + (n-1)x_{i,j} \leq n-2, \tag{2.4}$$

para $i, j \in \{2, \ldots, n\}$, $i \neq j$. Veamos que dichas condiciones son equivalentes al PAV. Probemos que un subtour no corresponde a una solución factible del problema entero mixto. Consideremos un subtour $t = \{(r_2, r_3), \ldots, (r_k, r_2)\}$ tal que no contiene al nodo 1 y con $3 \leq \text{card}(t) = k-1 \leq n-1$. Considerando las restricciones propuestas, para $i \in \{2, \ldots, k-1\}$ tenemos

$$\alpha_{r_i} - \alpha_{r_{i+1}} + (n-1)x_{r_i, r_{i+1}} \leq n-2,$$

y además

$$\alpha_{r_k} - \alpha_{r_2} + (n-1)x_{r_k, r_2} \leq n-2.$$

Como consideramos a t un subtour, todas las $x_{r_i, r_{i+1}} = 1 = x_{r_k, r_2}$, entonces se obtienen las desigualdades

$$\begin{aligned} \alpha_{r_i} - \alpha_{r_{i+1}} &\leq -1, && \text{para } i \in \{2, \ldots, k-1\}, \\ \alpha_{r_k} - \alpha_{r_2} &\leq -1. \end{aligned}$$

Si se suman las restricciones anteriores, obtenemos que $0 \leq -k+1$, lo cual no corresponde a una solución factible, ya que $k \geq 3$.

Probemos ahora que dado un tour factible, existen valores para las variables α's tales que satisfacen las desigualdades (2.4). Ahora, considerando a 1 como la ciudad inicial del tour, si el arco (i, j) pertenece a un tour ($x_{i,j} = 1$) e i es la k-ésima ciudad visitada, proponemos a $\alpha_i = k$. Esto nos lleva a que

$$\begin{aligned} \alpha_i - \alpha_j + (n-1)x_{i,j} &= k - (k+1) + n - 1 \\ &= n-2. \end{aligned}$$

Consideremos ahora que $x_{i,j} = 0$,

$$\begin{aligned} \alpha_i - \alpha_j + (n-1)x_{i,j} &= \alpha_i - \alpha_j \\ &\leq n-2, \end{aligned}$$

ya que en este caso $\alpha_i, \alpha_j \in \{1, \ldots, n-1\}$ pues 1 es la n-ésima y última ciudad en visitar, demostrando así que todo recorrido factible, cumple las restricciones.

El criterio con el cual compararemos a una solución con respecto a otra es por su costo, es decir, la suma de los costos de los arcos o aristas que conforman el tour. Con lo anterior, definimos a nuestra función objetivo como

$$z = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{i,j} x_{i,j}.$$

Por lo tanto, el problema general del agente viajero queda expresado de la forma

$$
\begin{aligned}
\text{Min} \quad & \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{i,j} x_{i,j}, & \\
\text{s.a} \quad & \sum_{\substack{j=1 \\ j \neq i}}^{n} x_{i,j} = 1, & i \in \{1, \ldots, n\}, \\
& \sum_{\substack{i=1 \\ i \neq j}}^{n} x_{i,j} = 1, & j \in \{1, \ldots, n\}, \\
\alpha_i - \alpha_j \; & +(n-1) x_{i,j} \leq n-2, & i, j \in \{2, \ldots, n\},\ i \neq j, \\
& x_{i,j} \in \{0,1\}, & i, j \in \{1, \ldots, n\},\ i \neq j.
\end{aligned}
$$

Es importante mencionar que no necesariamente tenemos simetría en los costos, es decir, no necesariamente tendremos que $c_{i,j} = c_{j,i}$. En este caso, la red $R = (X, A, c)$ debe ser dirigida. Un ejemplo práctico es considerar la entrega de paquetes, donde para ir de una dirección i a una dirección j en automóvil, la distancia de regreso no necesariamente es la misma. En la figura 2.2 podemos apreciar un ejemplo con vías de un sentido.

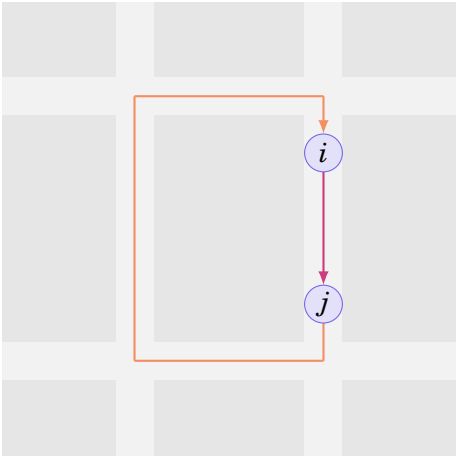


Figura 2.2: No necesariamente $c_{i,j} = c_{j,i}$.

2.1.2. El problema del mensajero

Otras variantes del problema del agente viajero surgen cuando no necesariamente queremos empezar y terminar en la misma ciudad. El problema del mensajero (PM) [1, págs. 157-159] busca el camino hamiltoniano de menor costo en una red R. De aquí surgen dos subvariantes, cuando se especifican los nodos de inicio y fin y cuando se escogen dichos nodos de manera arbitraria.

Veamos cómo resolver el PM sin especificar las ciudades de inicio y fin por medio del PAV. Sea $R = (X, A, c)$ una red dirigida completa y sin bucles, donde $X = \{1, \ldots, n\}$ y $c_{i,j} \geq 0$ para todo $(i, j) \in A$. Definimos a una nueva red $R' = (X', A', c')$ donde

$$X' = \{0, 1, \ldots, n\}, \qquad A' = \{(i,j) \in X' \times X' : i \neq j\},$$
$$c' : A' \to \mathbb{R}, \qquad c'(i,j) = c(i,j),$$
$$c'(i,0) = c'(0,i) = 0, \quad \text{para todo } i, j \in X,\ i \neq j.$$

Al resolver la instancia anterior del PAV, obtendremos un tour $t = \{(0,x_1), \ldots, (x_n,0)\}$ de costo $c(0,x_1) + \cdots + c(x_n,0)$. Nótese que si eliminamos los arcos los arcos $(0,x_1)$ y $(x_n,0)$, obtendremos el camino hamiltoniano de menor costo en la red G, ya que el costo de todos los arcos de la forma $(0,i)$ e $(i,0)$ es 0, para $i \in \{1,\ldots,n\}$.

2.2. El Problema de Asignación

Antes de ver un método para resolver el PAV, veremos una versión con menos restricciones, ya que resolverla resulta más fácil. El problema de asignación consiste en destinar tareas a ciertos trabajadores de la forma más eficiente para realizarlas [11, pág. 5]. La razón por la cuál se dice que el problema de asignación es una relajación del PAV, es porque el problema de asignación tiene en esencia las restricciones (2.1) y (2.2) y no las demás.

Así como el agente viajero, el problema de asignación puede ser representado a partir de una red o como un problema de programación lineal binario. Esta última forma resulta ser de nuestro interés, ya que un resultado teórico nos permite reducir el problema a uno de programación lineal. A lo largo de esta sección, veremos un ejemplo del problema de asignación, su planteamiento general, algunas de sus propiedades matemáticas y daremos un algoritmo para resolver el problema a partir de una gráfica bipartita. Para esta sección utilizaremos lo presentado en el libros *Assignment problems* de Bukard, Dell'Amico y Martello [11] e *Integer Programming* de Conforti, Cornuejols y Zambelli [12].

2.2.1. Planteamiento general

Supongamos que tenemos n tareas a realizar y contamos con n personas para ello. Cada persona puede resolver una sola tarea y todas las tareas deben de ser resueltas. Suponemos que no todas las personas tienen la misma capacidad de resolver una tarea en cuestión de costo, el cual puede representar tiempo, dinero, entre otros. El problema de asignación busca resolver todas las tareas de manera óptima en tiempo secuencial.

Definimos a las variables de decisión del problema. Sea $x_{i,j} \in \{0,1\}$ dada por

$$x_{i,j} = \begin{cases} 1, \text{ si se asigna la tarea } i \text{ a la persona } j, \\ 0, \text{ e.o.c.} \end{cases}$$

donde $i, j \in \{1, \ldots, n\}$. Para las restricciones, tenemos dos supuestos a considerar. El primero nos indica que cada persona se puede encargar de resolver una y sólo una tarea, entonces dada una persona $j \in \{1, \ldots, n\}$, tenemos que

$$\sum_{i=1}^{n} x_{i,j} = 1.$$

El segundo nos indica que todas las tareas deben de ser resueltas, por lo que se le debe de asignar cada una de ellas a una sola persona, es decir, dada $i \in \{1,...,n\}$,

$$\sum_{j=1}^{n} x_{i,j} = 1.$$

Ahora, definimos al peso de nuestra red como el costo $c_{i,j} \geq 0$ que implicaría asignarle a la persona j la tarea i. Nuestra función objetivo quedaría expresada como

$$z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j}$$

Por lo tanto, nuestro modelo queda de la siguiente forma

$$\begin{aligned}
\text{Min} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j} \\
\text{s.a} \quad & \sum_{i=1}^{n} x_{i,j} = 1, \qquad j \in \{1,...,n\}, \\
& \sum_{j=1}^{n} x_{i,j} = 1, \qquad i \in \{1,...,n\}, \\
& x_{i,j} \in \{0,1\}, \quad i,j \in \{1,...,n\}.
\end{aligned}$$

Notemos que el problema de asignación es un problema de programación lineal binario. El problema anterior puede ser representado con una red bipartita $(U \cup V, A, c)$, donde U es el conjunto de tareas, V es el conjunto de personas, $A = U \times V$ y finalmente $c : A \to \mathbb{R}$ es la función de costo. Los costos de los arcos pueden ser expresados en una matriz de $n \times n$. Definimos a la matriz $C \in M_{n\times n}(\mathbb{R})$ dada por

$$c_{i,j} = c(i,j), \quad \text{donde } i,j \in \{1,...,n\}.$$

Ejemplo 2.2. Mauricio, Santiago, Iker y Bruno deben de exponer una presentación de cuatro temas en su clase de física. Ellos quieren exponer los temas en el menor tiempo posible para salir lo antes posible de la clase. En la tabla 2.2, se da una estimación en minutos de cuanto les tomaría exponer cada tema a cada uno. Notemos que la información de la tabla 2.2 puede ser plasmada en una red.

Tabla 2.2: Los minutos que tardarían en resolver cada ejercicio.

	Integrantes			
Temas	Mauricio(1)	Santiago(2)	Iker(3)	Bruno(4)
1	20	21	15	17
2	17	16	20	19
3	11	29	19	22
4	13	12	30	19

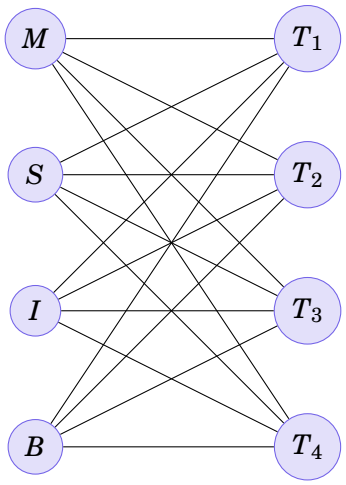


Figura 2.3: Gráfica de las tareas y los integrantes del equipo.

Con la tabla anterior podemos construir una matriz $C \in M_{4\times 4}(\mathbb{R})$, la cual contendrá en la entrada $c_{i,j}$ el tiempo promedio en el que el integrante i tardará en exponer el tema j.

$$C = \begin{pmatrix} 20 & 21 & 15 & 17 \\ 17 & 16 & 20 & 19 \\ 11 & 29 & 19 & 22 \\ 13 & 12 & 30 & 19 \end{pmatrix}.$$

2.2.2. El algoritmo húngaro

Como ya mencionamos anteriormente, la relajación PL del modelo lineal binario del problema de asignación tiene soluciones enteras. No es necesario considerar la restricción de que cada $x_{i,j} \in \{0,1\}$, ya que al resolver el problema lineal, se garantiza que se cumple. Al representar las condiciones anteriores en forma matricial ($Ax = b$), tenemos que A es una matriz totalmente unimodular y b es un vector de entradas enteras.

Definición 2.2. Sea $A \in M_{m\times n}(\mathbb{R})$ una matriz con coeficientes enteros, decimos que es una matriz **totalmente unimodular** si para cualquier submatriz cuadrada S de A, se tiene que $\det(S) \in \{1, 0, -1\}$.

Notemos que el planteamiento general del problema de asignación se puede reescribir en forma matricial como

$$\begin{aligned} \text{Min} \quad & c^t x, \\ \text{s.a} \quad & Ax = b, \\ & x \in \{0,1\}^{n^2}. \end{aligned}$$

donde $b = (1, \ldots, 1)^t \in \mathbb{R}^{2n}$, $c \in \mathbb{R}^{n^2}$ y $A \in M_{2n\times n^2}(\mathbb{R})$ conformada por ceros (representados por espacios vacíos) y unos definida como

$$A = \left(\begin{array}{ccc|ccc|c|ccc} 1 & \cdots & 1 & & & & & & & \\ & & & 1 & \cdots & 1 & & & & \\ & & & & & & \ddots & & & \\ & & & & & & & 1 & \cdots & 1 \\ \hline 1 & & & 1 & & & & 1 & & \\ & \ddots & & & \ddots & & \cdots & & \ddots & \\ & & 1 & & & 1 & & & & 1 \end{array}\right).$$

Observemos que x es el vector de componentes $x_{i,j}$. La matriz anterior, es una matriz totalmente unimodular [11, pág. 74]. Además, si a la matriz anterior le retiramos una fila cualquiera, la matriz resultante sería de rango completo, es decir, el rango de la matriz A es de $2n-1$ [13, pág. 172].

Teorema 2.3. *Sea $A \in M_{m\times n}(\mathbb{R})$ una matriz totalmente unimodular ($m \leq n$), $b \in \mathbb{Z}^m$ y $c \in \mathbb{R}^n$. El problema de programación lineal en forma estándar dado por*

$$\begin{array}{ll} \text{Min} & c^t x, \\ \text{s.a} & Ax = b, \\ & x \geq 0. \end{array}$$

donde $\{y \in \mathbb{R}^n : Ay = b, y \geq 0\}$ es una región no vacía tiene una solución óptima $x^* \in \mathbb{Z}^n$.

Demostración. Sin pérdida de generalidad, supongamos que A es de rango completo, es decir, $\text{rango}(A) = m$. En caso de que A sea de rango $p < m$, como $\{y \in \mathbb{R}^n : Ay = b, y \geq 0\}$ es una región no vacía, entonces el sistema tiene solución. Por el teorema de Rouché–Capelli podemos eliminar p filas con índices $\{l_1, \ldots, l_p\}$ de la matriz para que la matriz resultante sea de rango completo. Lo mismo hacemos para el vector b en los índices $\{l_1, \ldots, l_p\}$. El sistema resultante representaría la misma región.

Veamos que cualquier sbf tiene entradas enteras [14]. Recordemos que si x satisface las restricciones anteriores, entonces $x \in \{y \in \mathbb{R}^n : Ay = b, y \geq 0\}$. Si $\{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\}$ es un conjunto de índices tal que el conjunto $\{A_{j_1}, \ldots, A_{j_m}\}$ es una base de $\mathbb{R}^m$, entonces la matriz $B \in M_{m\times m}(\mathbb{R})$ dada por $B = [A_{j_1}|\cdots|A_{j_m}]$ es no singular. Recordemos que si $x \in \mathbb{R}^n$ es una sbf para el problema anterior, entonces para toda $l \in \{1, \ldots, n\} \setminus \{j_1, \ldots, j_m\}$ tenemos que $x_l = 0$ y para toda $k \in \{1, \ldots, m\}$, $x_{j_k} = (B^{-1}b)_k$. Por la regla de Crámer [15, pág. 72], tenemos que $(B^{-1}b)_k = \det(B_k)/\det(B)$, donde B_k es la matriz resultante de sustituir la fila k con el vector b en la matriz B. Como A es totalmente unimodular y B es una submatriz invertible de A, entonces $\det(B)$ es 1 o -1. Como B es totalmente unimodular y los enteros son cerrados por las operaciones de suma, resta y producto, entonces $\det(B_k) = \sum_{\sigma\in\mathcal{S}_n}^{n} \varepsilon(\sigma)\prod_{j=1}^{n}(B_k)_{j,\sigma(j)} \in \mathbb{Z}$ [15, págs. 19–20]. Así queda demostrado que toda sbf es entera y por lo tanto tiene un óptimo $x^* \in \mathbb{Z}^n$. □

El teorema anterior es una versión modificada del teorema de Hoffman y Kruskal [12]. Notemos que el problema de asignación satisface las hipótesis previas, por ello podemos asegurar que si relajamos la restricción $x \in \{0,1\}^{n^2}$ a $0 \leq x \leq 1$, entonces la solución óptima x^* para el problema será entera. Más aún, por la restricción $Ax = b$, si relajamos $0 \leq x \leq 1$ a $x \geq 0$, se garantiza que x tendrá únicamente entradas iguales a 0 o

1. Consideremos el problema de programación lineal dado por

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}\,x_{i,j}, \\ \text{s.a} \quad & \sum_{j=1}^{n} x_{i,j} = 1, \quad i \in \{1,\dots,n\}, \\ & \sum_{i=1}^{n} x_{i,j} = 1, \quad j \in \{1,\dots,n\}, \\ & x_{i,j} \geq 0, \quad i,j \in \{1,\dots,n\}. \end{aligned}$$

Entonces, el problema dual está dado por

$$\begin{aligned} \text{Max} \quad & \sum_{i=1}^{n} (u_i + v_i), \\ \text{s.a} \quad & u_i + v_j \leq c_{i,j}, \quad i,j \in \{1,\dots,n\}. \end{aligned}$$

De aquí podemos aplicar el teorema de holguras complementarias. Tenemos que dados x, u, v, la solución es óptima si y sólo si se satisface

$$x_{i,j}(c_{i,j} - u_i - v_j) = 0, \quad \forall i,j \in \{1,...,n\}.$$

Podemos definir a los coeficientes de costo reducido de la forma

$$\overline{c}_{i,j} := c_{i,j} - u_i - v_j, \quad \forall i,j \in \{1,...,n\}.$$

Con ayuda de los coeficientes de costo reducido y resolviendo el problema dual, se plantea el algoritmo húngaro, el cual determina una solución para el problema de asignación. El algoritmo consiste en 3 pasos. El primero llamado **preprocesamiento** es encontrar una solución parcial del primal y factible del dual. El segundo paso llamado **húngaro** consiste en verificar si la asignación es total o parcial, en caso de ser parcial, se procede al tercer paso. El tercer paso llamado **procedimiento alternante** tiene como objetivo encontrar una cadena o un camino aumentante para introducir una tarea no asignada.

Es importante remarcar que utilizaremos indexaciones para tratar a los elementos de U y V. Como $\text{card}(U) = \text{card}(V) = n$, podemos indexar a los conjuntos anteriores con los elementos del conjunto $\{1,\dots,n\}$. Por ello, haremos referencia a los elementos de U y V como el i-ésimo elemento de U o el j-ésimo elemento de V, para $i,j \in \{1,\dots,n\}$. Además, representaremos a las entradas del vector $x \in \mathbb{R}^{n^2}$ en la matriz $X \in M_{n\times n}(\mathbb{R})$. En la tabla 2.2 aparece la indexación del ejemplo 2.2.

Definimos a f y φ dados por

$$f(j) := \begin{cases} i & \text{si a la columna } j \text{ se asigna la fila } i \ (x_{i,j} = 1), \\ \infty & \text{e.o.c.} \end{cases}$$

$$\varphi(i) := \begin{cases} j & \text{si a la fila } i \text{ se asigna la columna } j \ (x_{i,j} = 1), \\ \infty & \text{e.o.c.} \end{cases}$$

Sea $X \in M_{n\times n}(\mathbb{R})$, dada por $x_{i,j} = 0$, para toda $i, j \in \{1, ..., n\}$. Dentro de esta matriz guardaremos las asignaciones de las tareas a las personas. La entrada $x_{i,j}$ de X, representa a la entrada $x_{n(i-1)+j}$ del vector x. Sea $\overline{U} = \varnothing$ el conjunto de tareas asignadas. Sean $u, v, f, \varphi \in \overline{\mathbb{R}}^n$ vectores tales $u_i, v_i = 0$ y $f_i, \varphi_i = \infty$ para toda $i \in \{1, ..., n\}$, que sus entradas son cero.

Preprocesamiento

1. Obtener el valor mínimo de cada renglón $i \in \{1, \ldots, n\}$ en la matriz C y guardar el número en la i-ésima entrada del vector u.

2. Para cada $j \in \{1, \ldots, n\}$, asignar los valores $v_j = \text{mín}\{c_{i,j} - u_i\}$.

3. Para todas las filas $i \in \{1, \ldots, n\}$, revisar el j-ésimo elemento de la fila. Si el elemento es cero, si i no está en el conjunto de tareas asignadas ($\overline{U}$) y si la j-ésima entrada del vector f es infinito, entonces introducimos a i a las tareas asignadas, $f(j) = i$ y $\varphi(i) = j$.

Al final del algoritmo de preprocesamiento, tendremos a la matriz de costos reducidos, la cual usaremos más adelante. Resolvamos el ejemplo 2.2, primero debemos de encontrar la solución factible del dual (u, v). Sea $C \in M_{4\times 4}(\mathbb{R})$ dada por

$$C = \begin{pmatrix} 20 & 21 & 15 & 17 \\ 17 & 16 & 20 & 19 \\ 11 & 29 & 19 & 22 \\ 13 & 12 & 30 & 19 \end{pmatrix}.$$

De aquí obtenemos a u.

$$\begin{aligned} u_1 &= \text{mín}\{20, 21, 15, 17\} = 15, \\ u_2 &= \text{mín}\{17, 16, 20, 19\} = 16, \\ u_3 &= \text{mín}\{11, 29, 19, 22\} = 11, \\ u_4 &= \text{mín}\{13, 12, 30, 19\} = 12. \end{aligned}$$

Ahora podemos obtener a v.

$$\begin{aligned} v_1 &= \text{mín}\{20 - 15, 17 - 16, 11 - 11, 13 - 12\} = 0, \\ v_2 &= \text{mín}\{21 - 15, 16 - 16, 29 - 11, 12 - 12\} = 0, \\ v_3 &= \text{mín}\{15 - 15, 20 - 16, 19 - 11, 30 - 12\} = 0, \\ v_4 &= \text{mín}\{17 - 15, 19 - 16, 22 - 11, 19 - 12\} = 2. \end{aligned}$$

Tabla 2.3: Algoritmo de preprocesamiento para el Ejemplo 2.2.

Índices				
i	j	$f(j)$	$\overline{c}_{ij}$	$\overline{U}$
1	1	∞	5	$\varnothing$
	2	∞	6	$\varnothing$
	3	∞	0	$\varnothing$
	4	∞	0	$\{1\}$
2	1	∞	1	$\{1\}$
	2	∞	0	$\{1\}$
	3	1	4	$\{1,2\}$
	4	∞	1	$\{1,2\}$
3	1	∞	0	$\{1,2\}$
	2	2	18	$\{1,2,3\}$
	3	1	8	$\{1,2,3\}$
	4	∞	9	$\{1,2,3\}$
4	1	3	1	$\{1,2,3\}$
	2	2	0	$\{1,2,3\}$
	3	1	18	$\{1,2,3\}$
	4	∞	5	$\{1,2,3\}$

Con la solución factible, podemos asignar los valores iniciales a los vectores f, φ y al conjunto $\overline{U}$. Aplicando el algoritmo, tenemos que $u = (15,16,11,12)^t$, $v = (0,0,0,2)^t$, $f = (3,2,1,\infty)^t$, $\varphi = (3,2,1,\infty)^t$ y $\overline{U} = \{1,2,3\}$. Dada una solución factible (u,v) del problema dual y los vectores f, φ, podemos definir una gráfica $G_0 = (U \cup V, A_0)$, donde A_0 es el acoplamiento de G obtenido en el preprocesamiento de la siguiente manera: $(i,j) \in A_0$ si y sólo si $\varphi(i) = j$, para $i, j \in \{1,\ldots,n\}$. Definimos a los siguientes conjuntos:

$$SU := \text{Los vértices examinados del conjunto } U,$$
$$SV := \text{Los vértices examinados del conjunto } V,$$
$$LV := \text{Los vértices etiquetados del conjunto } V.$$

Podemos incluir a los conjuntos SU, SV y LV en una gráfica con la notación definida en la figura 2.4. El siguiente algoritmo tomará un nodo $k \in U \setminus \overline{U}$ y encontrará una cadena o camino A_0-alternante enraizado en k. Las aristas o arcos de la gráfica son actualizadas con A_0. Este procedimiento se repite hasta que $U = \overline{U}$. Dos componentes esenciales para los otros dos pasos del algoritmo, son los vectores $\pi, \text{pred} \in \overline{\mathbb{R}}^n$, los cuales almacenaran los coeficientes de costo reducido más pequeños de cada nodo $j \in V$ con $i \in SU$ y los nodos para la cadena alternante respectivamente. El vector pred es construido a partir del vector π y de un nodo expuesto.

Consideraremos que una iteración es cada vez que escogemos un nodo no asignado $k \in U \setminus \overline{U}$. También consideramos que una subiteración del algoritmo se da cada vez que se actualicen los vectores pred y π. Para cada iteración y subiteración, se presenta el esquema de la figura 2.5. Los arcos de la gráfica, representan el acoplamiento actual. Es importante observar que se necesitará a solución dual (u,v) y a la matriz de costos

durante todo el procedimiento.

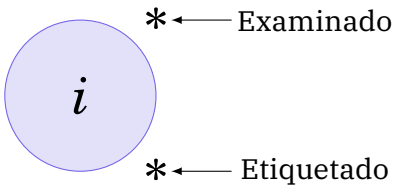


Figura 2.4: Notación para el algoritmo húngaro.

Húngaro

1. Si $U = \overline{U}$, FIN. En otro caso, ir al paso 2.
2. Sea $k \in U$ un nodo expuesto. Hacer $\text{pred} = (k, \ldots, k)$. Obtener $j \in V$ y atualizar pred por medio del procedimiento alternante.
3. Introducimos a k en $\overline{U}$. Actualizamos el acoplamiento, es decir, mientras que $i \neq k$, hacer $i = \text{pred}(j)$, $f(j) = i$, $h = \varphi(i)$, $\varphi(i) = j$ y $j = h$. Ir al paso 1.

Procedimiento alternante

1. Hacer $\pi = (\infty, \ldots, \infty)$ y $k = i$.
2. Marcar como examinado al nodo i. Para cada $j \in V$ tal que $\overline{c}_{i,j} < \pi(j)$, hacer $\text{pred}(j) = i$ y $\pi(j) = \overline{c}_{i,j}$ y si $\pi(j) = 0$, entonces etiquetar a j.
3. Si todos los nodos en V etiquetados ya fueron examinados, entonces se procede a hacer la actualización dual.
   - Tomar $\delta = \text{mín}\{\pi(j) : j \text{ es un vértice no etiquetado de } V\}$.
   - Para cada nodo $i \in U$ examinado, hacer $u_i = u_i + \delta$.
   - Para cada nodo $j \in V$ etiquetado, hacer $v_j = v_j - \delta$.
   - Para cada nodo $j \in V$ no etiquetado, hacer $\pi(j) = \pi(j) - \delta$. Si $\pi(j) = 0$, entonces etiquetar a j.
4. Sea $j \in V$ un nodo etiquetado pero no examinado. Marcar a j como examinado. Si j no está acoplado (es decir, $f(j) = \infty$), terminar el algoritmo y regresar a j como un nodo expuesto. En otro caso, hacer i igual al nodo acoplado con j (es decir, $i = f(j)$) e ir al paso 2.

Notemos que el procedimiento alternante buscará un nodo expuesto en V para encontrar una cadena o camino aumentante por medio de los coeficientes de costo reducido y una cadena de nodos entre U y V. En caso de no encontrar dicho nodo con la solución dual actual, se realiza una actualización dual al determinar una $\delta > 0$ para extender al conjunto de nodos sobre los cuales se puede aumentar la cadena.

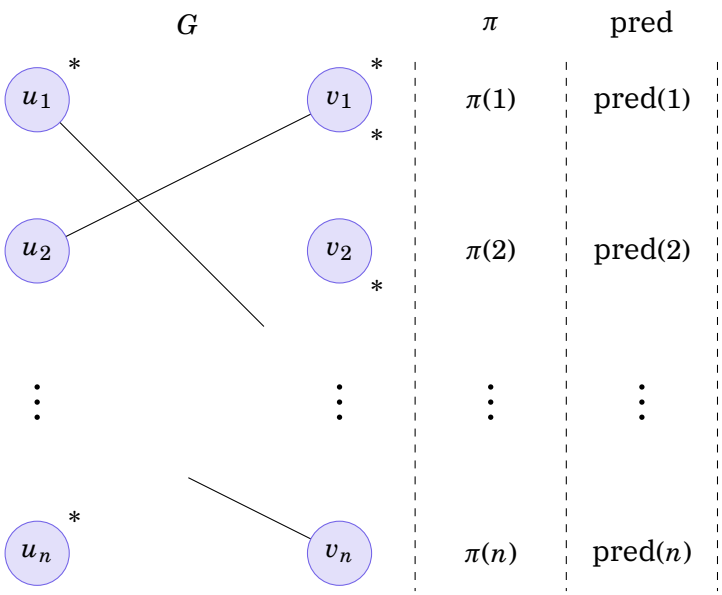


Figura 2.5: Esquema para cada subiteración.

Proposición 2.4. *La actualización dual del algoritmo Húngaro es tal que*

1. $\pi_j = \min\{\overline{c}_{i,j} : i \in SU\}$ *para toda* $j \in V$ *está bien definido;*
2. *los arcos de la red actual que conecten pares de nodos escaneado-etiquetado o no etiquetados mantienen un costo reducido de cero;*
3. *se encontrará una cadena aumentante;*
4. *los coeficientes de costo reducido son no negativos.*

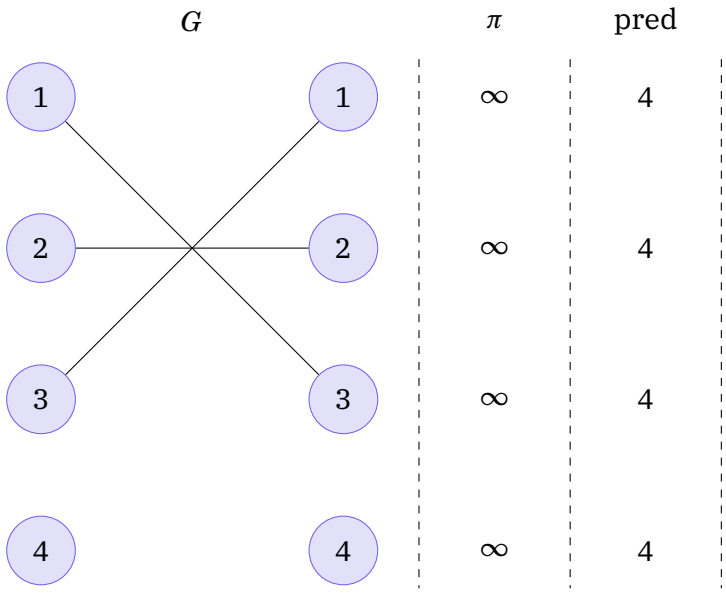
Demostración. Al inicio del algoritmo $\delta = \min\{\pi_j : j \in V \setminus LV, i \in SU\}$, donde $\pi_j = \overline{c}_{i,j}$.

1. Al hacer la actualización, $V \setminus LV \neq \varnothing$, ya que para entrar al procedimiento alternante, no debe haber un acoplamiento perfecto para G, entonces no se etiquetará a todo nodo en v. De lo anterior, al llegar a la actualización, debe existir $i \in SU$ y $j \in V \setminus LV$. Al actualizar las variables duales, obtendremos el coeficiente de costo reducido tal que $\pi_j = \overline{c}_{i,j}$ fue reducido en δ unidades, ya que u_i fue actualizado por $u_i + \delta$. Por la definición de δ, se concluye que $\overline{c}_{i,j} \geq 0$ para $j \in V \setminus LV$ y $i \in SU$.
2. Notemos que $(i,j) \in A_0$ si y sólo si $\overline{c}_{i,j} = 0$. Suponiendo que (i,j) es tal que $i \in SU$ y $j \in LV$, entonces se redefine a $0 = \overline{c}_{i,j} = \overline{c}_{i,j} + \delta - \delta = 0$. En caso de que $i \in U \setminus SU$ y $j \in V \setminus LV$, $\overline{c}_{i,j}$ no es actualizado.
3. Si $j \in V \setminus LV$, entonces se redefine a π_j como $\pi_j - \delta$. Por la definición de δ, al menos uno nodo $j \in V \setminus LV$ pasará a tener costo reducido 0, por lo que a partir del vector pred se puede encontrar una cadena aumentante.
4. De lo anterior, falta analizar el caso donde $i \in U \setminus SU$ y $j \in LV$. Aquí $\overline{c}_{i,j}$ aumenta δ unidades, ya que v_j es sustituido por $v_j - \delta$, por lo que $\overline{c}_{i,j} \geq 0$.

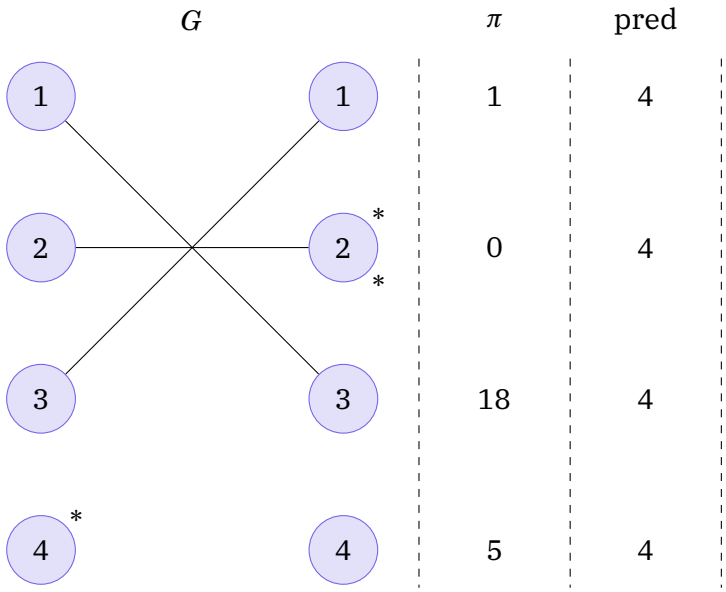
□

La proposición anterior garantiza que el algoritmo húngaro resuelve el problema de asignación. Regresando al ejemplo de esta sección, tenemos que el único nodo expuesto es 4, por lo que $k = 4$. Los esquemas quedarían como se muestra a continuación.

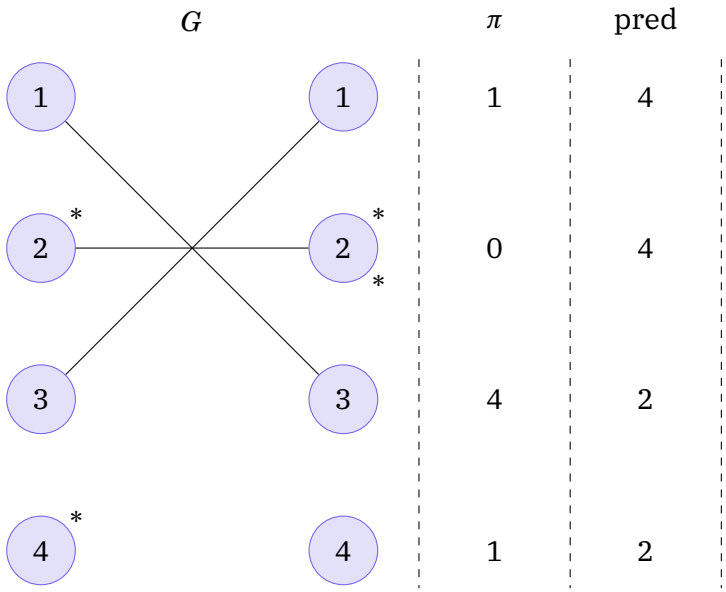
Iteración 1: $k = 4$, planteamos la gráfica a partir del segundo paso del algoritmo húngaro y del primer paso del procedimiento alternante.



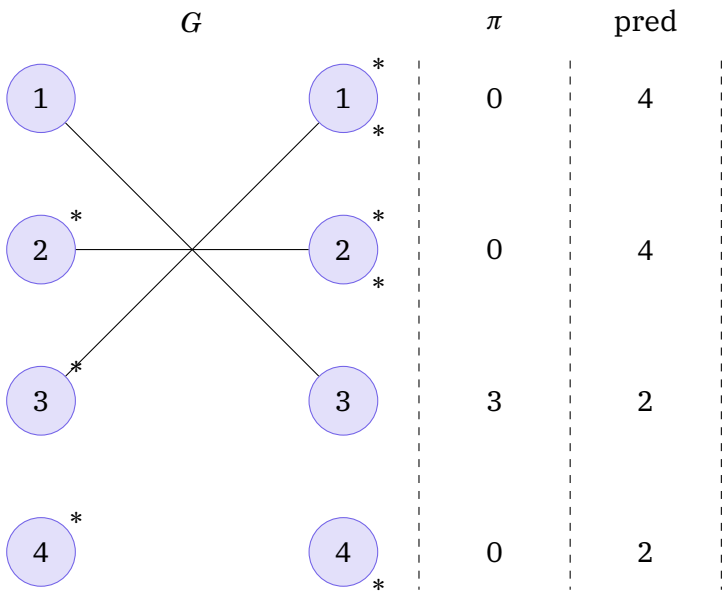
Subiteración 1.1: $i = 4$ y recordemos que la solución dual actual es $u = (15, 16, 11, 12)^t$ y $v = (0, 0, 0, 2)^t$. El paso 2 nos indica que marquemos al nodo 4 de la izquierda. Actualizamos al vector $\pi = (1, 0, 18, 5)^t$, pues cada entrada de la cuarta fila de $\overline{C}$ resulta ser finita. Etiquetamos a 2 de V. Saltamos el paso 3. El paso 4 nos indica que lo marquemos como examinado y que hagamos $i = 2$, ya que $\text{pred}(2) = 2$.



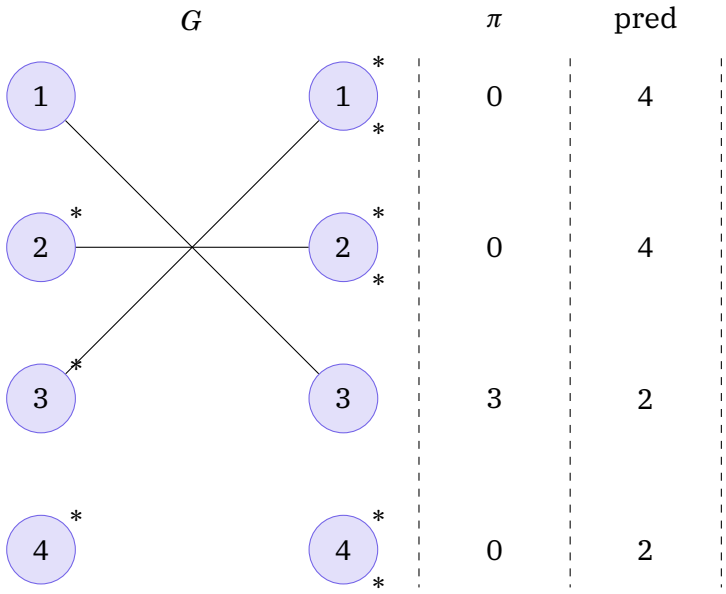
Subiteración 1.2: $i = 2$. Marcamos como examinado al nodo 2 de U. Notemos que $\overline{c}_{2,3} = 4 < 18$ y $\overline{c}_{2,4} = 1 < 5$, hacemos $\pi(3) = 4$, $\pi(4) = 1$, $\text{pred}(3) = 2$ y $\text{pred}(4) = 2$. No se etiquetan a nodos en V.



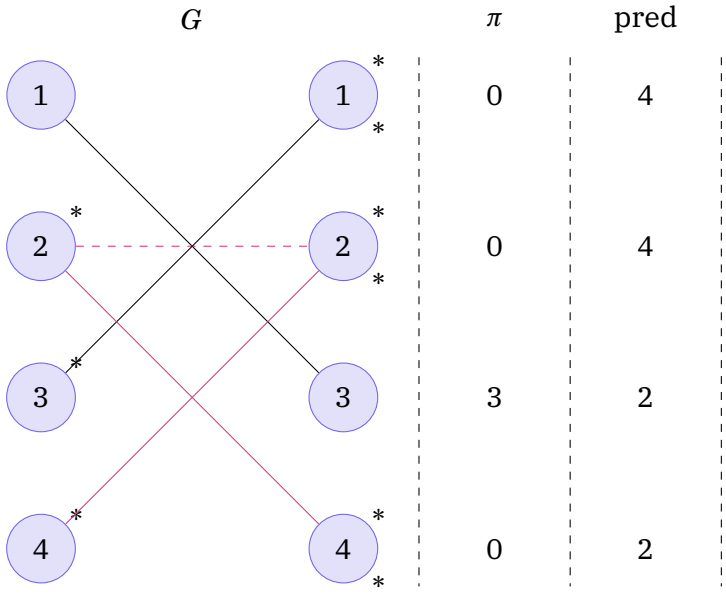
Subiteración 1.3: $i = 2$. Se actualiza la solución dual ya que todos los nodos etiquetados de V han sido examinados. Tomamos $\delta = \text{mín}\{1,4\} = 1$. Recodemos que la solución dual es $u = (15,16,11,12)^t$ y $v = (0,0,0,2)^t$. Los nodos examinados de U son 2 y 4, entonces $u = (15,16+1,11,12+1)^t$. El único nodo examinado de V es el 2, por lo que $v = (0,0-1,0,2)^t$. Por lo tanto, se tienen los vectores duales $u = (15,17,11,13)^t$ y $v = (0,-1,0,2)^t$. Los nodos no etiquetados de V son 1, 3 y 4, entonces $\pi = (1-1,0,4-1,1-1)^t = (0,0,3,0)^t$ y etiquetamos a 1 y 4 de V. Tomamos a $j = 1$ de V pues está etiquetado pero no examinado. Lo marcamos como examinado. Hacemos $i = 3$ y marcamos como examinado a i.



Subiteración 1.4: $i = 3$. No hay actualizaciones en π ni en los vectores duales. Tomamos a $j = 4$ en V. Marcamos a j como examinado. Hemos encontrado un nodo expuesto.



Para este punto, ya tenemos un nodo expuesto en V que fue examinado y etiquetado, por lo que ese nodo será el otro extremo del acoplamiento. Con ayuda del vector pred, podemos ver el nuevo acoplamiento. El paso 3 del algoritmo húngaro se muestra en los arcos de color fucsia.



Como el acoplamiento es perfecto, hemos terminado con el algoritmo húngaro. Podemos decir que el tema 1 lo debe de exponer Iker, el tema 2 lo debe de exponer Bruno, el tema 3 lo debe de exponer Mauricio y el tema 4 lo debe de exponer Santiago. La suma de los tiempos promedio en el que tardarán en exponer los temas, es de 57 minutos.

2.3. Algoritmo de ramificación y acotamiento

Recordemos que nuestro objetivo es resolver el PAV. El método que utilizaremos para resolverlo es conocido como algoritmo de ramificación y acotamiento o simplemente ramificación y acotamiento. El primer algoritmo de este tipo para resolver el PAV fue publicado en 1963 por Little, Murty, Sweeney y col. [16]. Dada una instancia del PAV, utilizaremos el algoritmo húngaro para resolver varios problemas de asignación hasta llegar al óptimo del PAV.

A continuación, lo que estudiaremos en esta sección. Primero, veremos un ejemplo ilustrativo de ramificación y acotamiento para un problema lineal entera en $\mathbb{R}^2$, pues es posible ver con gráficas el procedimiento. Posteriormente veremos cómo detectar circuitos que se vayan formando en la gráfica, con la finalidad de prevenir la formación de subtours durante el procedimiento. Finalmente, veremos el algoritmo de ramificación y acotamiento para el PAV y resolveremos el ejemplo 2.1.

2.3.1. Un ejemplo de programación entera

Sea x^* una solución óptima de un problema lineal entero. Notemos que si el óptimo $\widehat{x}$ para la versión relajada PL de un problema lineal entero satisface que todas sus entradas son enteras, entonces también es el óptimo del problema lineal entero, es decir, $z(x^*) = z(\widehat{x})$. En caso de que el óptimo de la relajación PL tenga al menos una entrada no entera, entonces $\widehat{z} = z(\widehat{x})$ es cota para $z^* = z(x^*)$. Si la dirección de optimización es maximizar, entonces $z^* \geq \widehat{z}$, en el otro caso, $z^* \leq \widehat{z}$. También es importante destacar que para cualquier x una solución factible del problema lineal entero, entonces $z(x) \leq z^*$ si queremos maximizar o $z(x) \geq z^*$ si queremos minimizar.

Ejemplo 2.3. Consideremos el problema lineal entero dado por

$$\begin{aligned} \text{Max} \quad & z = 6x_1 + 5x_2, \\ \text{s. a} \quad & 11x_1 + 18x_2 \leq 61, \\ & 3x_1 + 2x_2 \leq 13, \\ & x_1, \quad x_2 \geq 0, \\ & x_1, \quad x_2 \in \mathbb{Z}. \end{aligned}$$

En la figura 2.6, se muestran los puntos factibles del problema anterior, así como su relajación PL. Al ver la gráfica, no es claro cuál es la solución al problema entero. Llamemos a la relajación PL del problema anterior como problema #1. Al resolver el problema #1 por medio del algoritmo simplex [13, pág. 82], obtenemos que la solución óptima es $\widehat{x} = (7/2, 5/4)^t$ con $z(\widehat{x}) = 27.25$. Notemos que ninguna de las dos entradas del vector es entera, por lo que no es solución del problema lineal entero.

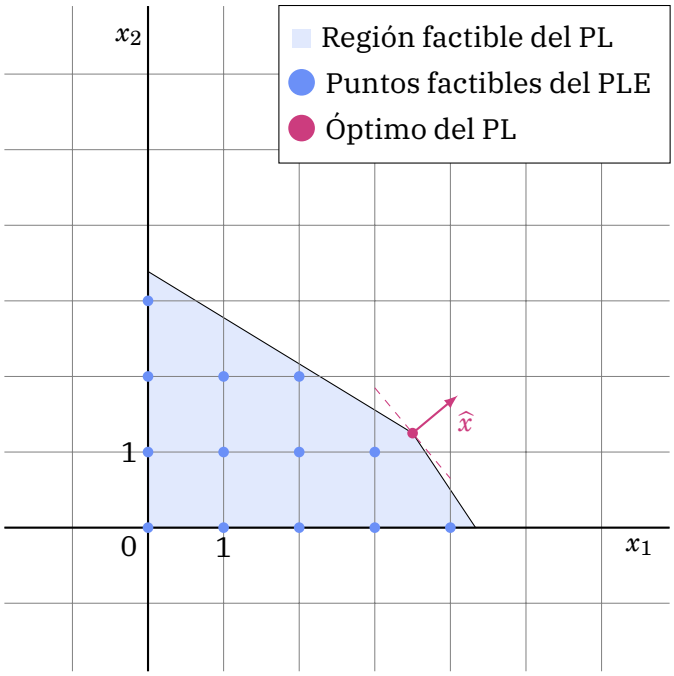


Figura 2.6: Gráfica del ejemplo 2.3.

Dada la solución óptima del problema #1, podemos aprovechar que la segunda entrada es 5/4, por lo que podemos ver qué pasa en los casos donde $x_2 \leq 1$ y $x_2 \geq 2$. Dichas regiones contienen a todas las soluciones enteras del problema original. Así, definimos dos nuevos problemas lineales.

Problema #2:

$$\begin{aligned} \text{Max} \quad & z = 6x_1 + 5x_2, \\ \text{s.a} \quad & 11x_1 + 18x_2 \leq 61, \\ & 3x_1 + 2x_2 \leq 13, \\ & x_2 \leq 1, \\ & x_1, \quad x_2 \geq 0. \end{aligned}$$

Problema #3:

$$\begin{aligned} \text{Max} \quad & z = 6x_1 + 5x_2, \\ \text{s. a} \quad & 11x_1 + 18x_2 \leq 61, \\ & 3x_1 + 2x_2 \leq 13, \\ & x_2 \geq 2, \\ & x_1, \quad x_2 \geq 0. \end{aligned}$$

Al proceso anterior, se le llama **ramificación**, ya que la unión de las dos regiones factibles de los problemas #2 y #3, contiene a todos los puntos factibles del #1. La solución óptima del #2 es $(11/3, 1)^t$ con $z(11/3, 1) = 27$, mientras que el #3 tiene solución óptima a $(25/11, 2)^t$ con $z(25/11, 2) = 23.\overline{63}$. La solución óptima de ambos problemas tienen un número no entero en la entrada x_1. Como el óptimo del #2 tiene valor 27 en la función objetivo, entonces posiblemente sea más prometedor ramificar ese problema.

Del problema #2, se definen a los #4 y #5, los cuales tienen el mismo planteamiento con las restricciones adicionales $x_1 \leq 3$ y $x_1 \geq 4$ respectivamente. El óptimo del #4 es $(3, 1)^t$ con $z(3, 1) = 23$ y el óptimo del #5 es $(4, 1/2)^t$ con $z(1, 1/2) = 26.5$. Como el óptimo del problema #4 ya es entero, podemos decir que $(3, 1)$ es un **candidato** para el óptimo del problema original y no es necesario seguir ramificando a partir de ese problema. Notemos que podemos ramificar a los #3 y #5. Como el óptimo del #5 tiene como valor

de la función objetivo $26.5 > 23.\overline{63}$, entonces podemos ramificarlo.

Del problema #5, se definen a los #6 y #7, los cuales tienen el mismo planteamiento con las restricciones adicionales $x_2 \leq 0$ y $x_2 \geq 1$ respectivamente. Observemos que la región definida por las restricciones del #7 es vacía, por lo que denotamos al problema de **no factible**. El óptimo del #6 es $(13/3, 0)^t$ con $z(13/3, 0) = 26.5$. Para este punto, podemos ramificar por el #3 y #6. Como el óptimo del #6 tiene como valor de la función objetivo $26 > 23.\overline{63}$, entonces podemos ramificarlo.

Del problema #6, se definen a los #8 y #9, los cuales tienen el mismo planteamiento con las restricciones adicionales $x_1 \leq 4$ y $x_1 \geq 5$ respectivamente. Observemos que la región definida por las restricciones del #9 es vacía, por lo que es un problema no factible. El óptimo del #8 es $(4, 0)^t$ con $z(4, 0) = 24$. Notemos que el óptimo del #8 ya es entero, por lo que es un nodo candidato. Además, $z(4, 0) = 24$ es mayor que $z(25/11, 2) = 23.\overline{63}$, por lo que podemos considerar al #3 como **no prometedor**. Cuando etiquetamos a los nodos como *no prometedores* o *no factibles*, se acota la región de búsqueda, lo que se conoce como **acotamiento**. Podemos observar que el óptimo es $(4, 0)^t$ con $z(4, 0) = 24$.

La unión de las regiones de los problemas que etiquetamos de candidatos y no prometedores, contiene a todos los puntos factibles del #1. En la figura 2.8 se encuentra la gráfica de las últimas regiones definidas en el proceso. Todo el procedimiento anterior, se puede plasmar en una estructura de árbol como se muestra en la figura 2.7. Al nodo asociado al problema #k, se le dice nodo #k. Al valor óptimo de la función objetivo del problema lineal #k, se le conoce como la **cota del nodo** k.

A los nodos 1, 2, 5 y 6 del árbol, se les dice nodos **no terminales**, ya que cada uno de ellos tienen ramificaciones. A los nodos 7 y 9 se les dice no factibles, ya que las regiones sobre la cual están definidos sus problemas lineales asociados, es una región vacía. Al nodo 3 se le dice nodo no prometedor, pues durante la ramificación se encontró una solución factible para el problema lineal entero cuyo valor óptimo era mayor que el óptimo del nodo 3, entonces no era necesario seguir ramificando por ese nodo. A los nodos 4 y 8 se les dice nodos candidatos, pues son los óptimos de los problemas lineales asociados a los nodos son vectores con entradas enteras, por lo que posiblemente son el óptimo del problema lineal entero. Los nodos no factibles, no prometedores y candidatos, son llamados nodos **terminales**.

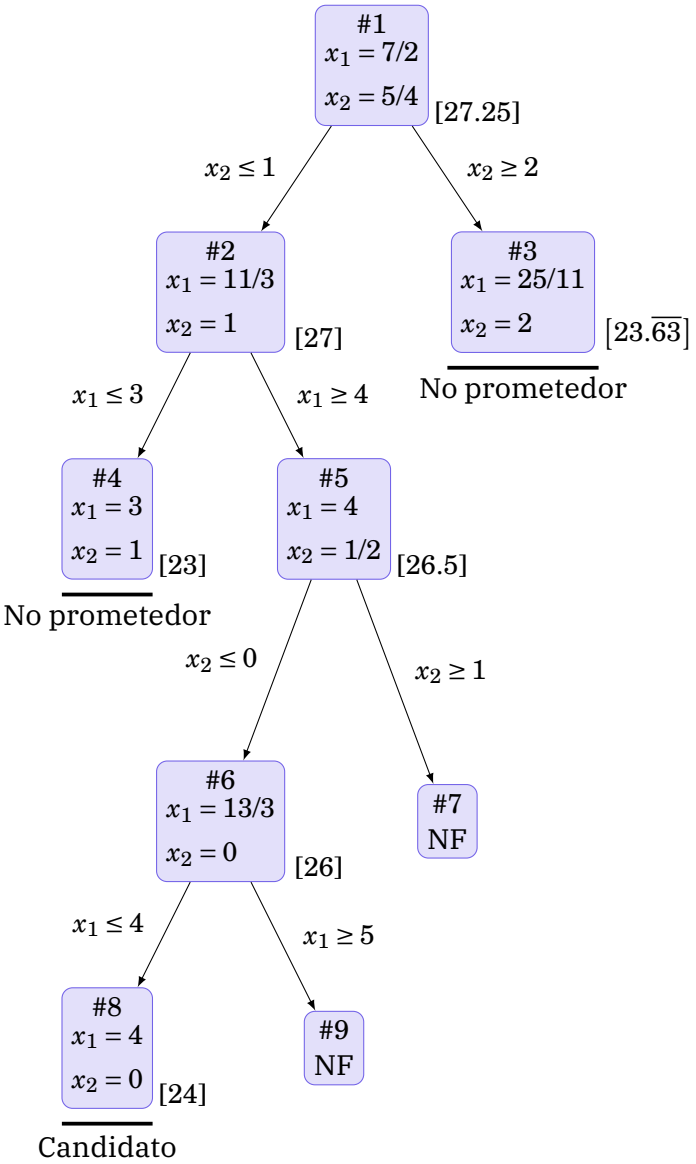


Figura 2.7: El árbol del algoritmo de ramificación y acotamiento para el ejemplo 2.3.

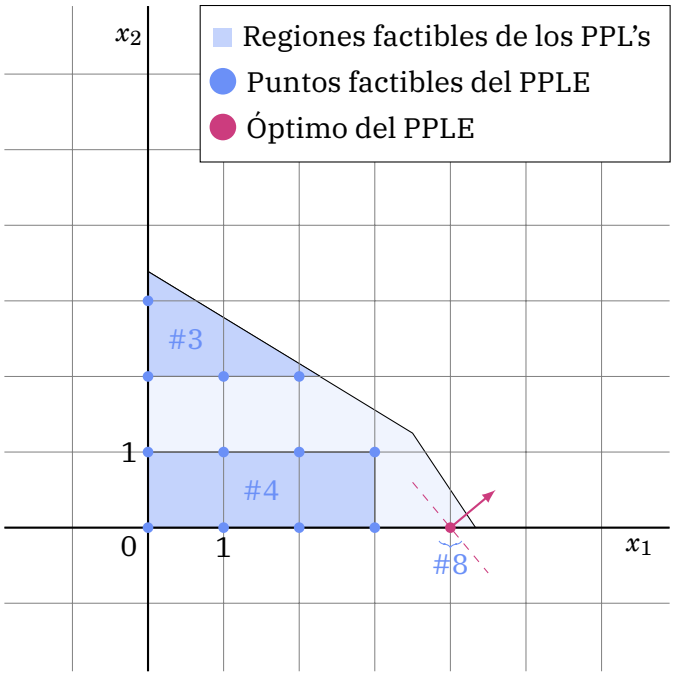


Figura 2.8: Las regiones definidas por los nodos terminales.

2.3.2. Detección de circuitos

Antes de adentrarnos al algoritmo de ramificación y acotamiento para resolver el PAV, es necesario que sepamos detectar circuitos cuando estemos agregando arcos a una digráfica parcial de G, ya que los no hamiltonianos llevarán a nodos no factibles. Consideremos a $G = (X, A)$ una digráfica completa y sin bucles, donde $X = \{1, \ldots, n\}$. A partir de G, queremos construir una digráfica parcial $G_n = (X, A_n)$ desde la gráfica $G_0 = (X, \varnothing)$. Bajo cierto criterio, iremos construyendo conjuntos A_i's donde $A_i = A_{i-1} \cup a$ hasta formar un circuito hamiltoniano $A_n \subseteq A$, para $i \in \{1, \ldots, n\}$ y $a \in A$. Entonces, necesitamos un algoritmo que verifique si A_i contiene circuitos para comprobar si la construcción se realiza de manera correcta. ¿Cómo detectamos un ciclo bajo esta premisa?

Sabemos que si a una digráfica $G = (X, A)$ se le agrega una arista a, pueden suceder dos cosas: 1) el número de componentes conexas disminuye en una unidad si la arista a une a dos componentes conexas y a no pertenece a algún ciclo; 2) el número de componentes conexas se mantiene igual y a pertenece a un ciclo de G [17, pág. 3]. Entonces, podemos contar el número de componentes conexas que tenga la gráfica $G_i = (X, A_i)$, para $i \in \{1, \ldots, n\}$. Notemos que en cada iteración A_i debería de tener exactamente $n - i$ componentes conexas. Dicho lo anterior, necesitamos un algoritmo que cuente componentes conexas.

En síntesis, el algoritmo va a tomar nodos de X e irá recorriendo a sus vecinos, a los vecinos de sus vecinos y así sucesivamente hasta no tener más nodos por recorrer bajo ese método. De esa forma, irá contando las componentes conexas. Al final, sólo restará saber si las componentes conexas de la gráfica G_i es igual a $n - i$. El algoritmo inicia con un contador de *componentes conexas* $c = 0$, el conjunto de *nodos no examinados* $N = X$ y

el conjunto de *nodos examinados* $E = \varnothing$.

Algoritmo de componentes conexas

1. Tomar $i \in N$ y hacer $E = E \cup \{i\}$.
2. Tomar $i \in E$ y actualizar $E = (\Gamma(i) \cap N) \cup (E \setminus \{i\})$ y $N = N \setminus \{i\}$. Si $E \neq \varnothing$, repetir el paso. Si $E = \varnothing$, Ir al paso 3.
3. Hacer $c = c+1$. Si $N \neq \varnothing$, entonces ir al paso 1. Si $N \neq \varnothing$, tenemos que c es la cantidad de componentes conexas.

Si introducimos un nodo de N a E, empezaríamos a analizar alguna componente conexa. Durante la ejecución del algoritmo, E iremos retirando a los nodos que vaya examinando para introducir a sus vecinos no examinados. Eventualmente se terminarían eliminando a todos los elementos de E. Para ese punto, se sabe que todos los nodos que se encontraron al repetir el paso 2, pertenecían a una sola componente conexa, por lo que se agrega una unidad a c. Este proceso se repite hasta que todos los nodos hayan sido examinados. Con la siguiente proposición, queda justificado el algoritmo.

Proposición 2.5. *Sea $G = (X, A)$ una digráfica completa y sin bucles. Si $G_i = (X, A_i)$ es una gráfica construida con ayuda del algoritmo anterior, entonces G_i tendría $n - i$ componentes conexas y no contiene ciclos, para $i \in \{1, \ldots, n\}$.*

Demostración. Procedemos por inducción sobre el conjunto $\{1, \ldots, n-1\}$.
Base inductiva: Sea $i = 1$, entonces se agregó a algún arco $a_1 = (i, j) \in A$. Al aplicar el paso 2 para $k \in X \setminus \{i, j\}$, iríamos directamente al paso 3, pues $\Gamma(k) = \varnothing$. Si $k \in \{i, j\}$, se repite una vez el paso 2. Con lo anterior, tenemos $n - 1$ componentes conexas. Además, con sólo un arco, la única forma de tener un ciclo es por medio de un bucle, lo cual contradice la hipótesis, por lo que G_1 es acíclica.

Hipótesis de inducción: Supongamos que para alguna $i \in \{1, \ldots, n\}$, la gráfica $G_i = (X, A_i)$ es acíclica y tiene $n - i$ componentes conexas.

Paso inductivo: p.d. $G_{i+1} = (X, A_{i+1})$ es acíclica y tiene $n - (i+1)$ componentes conexas. Sea $a_{i+1} \in A \setminus A_i$ la arista que se propone para construir a $A_{i+1} = A_i \cup \{a_{i+1}\}$. Procedemos por casos. En caso de que no disminuya en una unidad el número de componentes conexas de G_i a G_{i+1}, es porque a_{i+1} genera un ciclo en alguna componente conexa X_k, ya que ambos extremos de a_{i+1} se encuentran en X_k. Por lo tanto, se rechaza al candidato. En caso de que disminuya en una unidad el número de componentes conexas de G_i a G_{i+1}, es porque a_{i+1} tenía sus extremos en distintas componentes conexas. Al ejecutar el algoritmo anterior, tendríamos que la cantidad de componentes conexas es $n - (i+1)$ y no tenemos ciclos en la gráfica. □

Consideremos al ejemplo 2.1 y supongamos que hemos construido hasta el momento a $A_3 = \{(1,2), (2,3), (4,6)\}$. Notemos que $G_3 = (X, A_3)$ es una gráfica con tres componentes conexas ($\{1,2,3\}, \{4,6\}, \{5\}$) y sin circuitos. Supongamos que el criterio que estamos utilizando para construir a A_4 nos indica que $A_4 = A_3 \cup \{(3,1)\}$. Comenzamos con $N = \{1,2,3,4,5,6\}$, $E = \varnothing$ y $c = 0$.

Tabla 2.4: Ejemplo de detección de componentes conexas.

Paso	N	E	c
1	$\{1,2,3,4,5,6\}$	$\{1\}$	0
2	$\{2,3,4,5,6\}$	$\{2,3\}$	0
2	$\{3,4,5,6\}$	$\{3\}$	0
2	$\{4,5,6\}$	$\varnothing$	0
3	$\{4,5,6\}$	$\varnothing$	1
1	$\{4,5,6\}$	$\{4\}$	1
2	$\{5,6\}$	$\{6\}$	1
2	$\{5\}$	$\varnothing$	1
3	$\{5\}$	$\varnothing$	2
1	$\{5\}$	$\{5\}$	2
2	$\varnothing$	$\varnothing$	2
3	$\varnothing$	$\varnothing$	3

De la tabla 2.4, sabemos que hay 3 componentes conexas en $(X, A_3 \cup \{(3,1)\})$, por lo que rechazamos a $(3,1)$. Suponiendo que tras rechazar a $(3,1)$, el criterio ahora sugiere meter a $(5,1)$, veríamos que la cantidad de componentes conexas es 2, por lo que podríamos aceptar a $(5,1)$, ya que no genera ciclos.

Finalmente, para verificar que el ciclo hamiltoniano que se va formando es un circuito, debemos comprobar que cada arco que se agregue no comparta extremos iniciales o extremos finales con algún otro arco de la digráfica parcial. Lo anterior no es necesario incluirlo en el algoritmo de componentes conexas y veremos la razón más adelante. Con lo anterior, tenemos lo necesario para detectar a los nodos no factibles del algoritmo de ramificación y acotamiento presentado en la siguiente subsección.

2.3.3. Ramificación y acotamiento para el PAV

Consideremos al PAV de n ciudades. Definimos a C como la matriz $c_{i,j} = c(i,j)$ (el costo de ir de la ciudad i a la ciudad j) para $(i,j) \in A$, y $c_{i,i} = \infty$ para $i \in \{1,\dots,n\}$. Definir los costos de la diagonal como infinitos, previene que elijamos esas alternativas al resolver un problema de asignación. Sin embargo, no aseguramos que se prevenga la formación de subtours, pero obtendremos una cota para el problema del agente viajero. A partir de este punto, denotaremos a las cotas del nodo k como w_k.

En el primer nodo o problema #1, se debe resolver el problema de asignación para la matriz C y obtener la matriz de costos reducidos $\overline{C}^{(1)}$ (el subíndice denota el número del problema). Con la matriz $\overline{C}^{(1)}$, la ramificación se realiza a partir de un conjunto de penalizaciones, el cual es propuesto por Little, Murty, Sweeney y col. [16].

Definición 2.6. Dada una matriz $\overline{C} \in M_{n\times n}(\overline{\mathbb{R}})$ de coeficientes de costo reducido. Si $\overline{c}_{i,j} = 0$, definimos a la **penalización del arco** (i,j) como

$$\theta(i,j) := \min\left\{\overline{c}_{k,j} : k \in \{1,\dots,n\},\ k \neq i\right\} + \min\left\{\overline{c}_{i,l} : l \in \{1,\dots,n\},\ l \neq j\right\}.$$

Definimos al conjunto de **penalizaciones** de $\overline{C}$ como

$$\theta := \{\theta(i,j) < \infty : \overline{c}_{i,j} = 0,\ i,j \in \{1,\dots,n\}\}.$$

Entonces, dado el conjunto de penalizaciones de una matriz de costos reducidos, ramificamos sobre la variable $x_{i,j}$ tal que $\theta(i,j) = \max\theta$ bajo el criterio del nodo más prometedor (el mismo criterio de la figura 2.9). Notemos que al ser un problema binario, las nuevas restricciones serían $x_{i,j} = 1$ o $x_{i,j} = 0$ para cada rama. Lo anterior se divide en dos nodos tales que uno "contiene" todos los tours tales que tienen al arco (i,j) y el otro contiene a los que no tienen al arco (i,j). En caso de tener $x_{i,j} = 1$, podemos hacer $x_{j,i} = 0$ para evitar la formación de subtours por medio de esa variable. Al ramificar de la forma anterior, no se descartan soluciones factibles.

Dado que en el algoritmo húngaro no podemos asignar un valor a las variables de decisión de forma directa, modificaremos la matriz de costos C para que las variables x's tomen los valores deseados. Supongamos que se ramificó a algún nodo $\#p$ y se obtuvieron a los nuevos nodos $\#q$ y $\#q+1$, donde en $\#q$ necesitamos que $x_{i,j} = 0$ y que en $\#q+1$, $x_{i,j} = 1$ y $x_{j,i} = 0$. Ya hemos dicho anteriormente que al definir un costo infinito, forzamos a que la variable sea 0, por lo que podemos definir $\overline{C}^{(q)}$ igual a $\overline{C}^{(p)}$ excepto en la entrada (i,j), la cual tendrá valor infinito. Si consideramos a $\overline{C}^{(q+1)}$ igual a la matriz $C^{(p)}$, definimos sus entradas $(j,i),(i,l),(k,j)$ (con $l \in \{1,...,n\}$, $l \neq j$ y $k \in \{1,...,n\}$, $k \neq i$) igual a infinito y su entrada (i,j) igual a cero, entonces forzamos a que $x_{i,j} = 1$ y $x_{j,i} = 0$.

Por la forma en la que se define al conjunto θ y las modificaciones que se hacen a una matriz $\overline{C}$ para que $x_{i,j} = 1$, no obtendremos soluciones no factibles para el problema de asignación, es decir, soluciones que no satisfagan las ecuaciones (2.1) y (2.2). Si durante la ramificación, en algún nodo obtenemos circuitos no hamiltonianos formados por las variables $x's$ con valor 1, entonces etiquetaremos a ese nodo como no factible.

Para encontrar circuitos no hamiltonianos, es suficiente aplicar el algoritmo de componentes conexas. Por las modificaciones que se realizan a las matrices de costo reducido, no es posible tener dos variables $x_{i,j}, x_{k,j} = 1$, ya si al ramificar primero a $x_{i,j} = 1$, tendríamos que $\overline{C}^{q}_{k,j} = \infty$, lo que haría imposible esa ramificación. Por una justificación similar, también llegamos a que no es posible tener dos variables $x_{i,j}, x_{i,l} = 1$. Por la argumentación anterior, no es posible tener la formación de árboles ramificados durante la ejecución del algoritmo, sólo debemos evitar la formación de circuitos.

Algoritmo de ramificación y acotamiento para el PAV

1. Consideremos C la matriz de costos original, $q = 2$, X la matriz de $n \times n$ con coeficientes 0 y $z_0 = \infty$ el costo de la mejor solución factible actual. Empezamos a desarrollar nuestro árbol con el nodo #1. Resolvemos el problema de asignación para C y obtenemos a la matriz de coeficientes de costo reducido $\overline{C}^{(1)}$ junto con la matriz de asignación $X^{(1)}$. Hacemos $w_1 = \sum_{m=1}^{n} u_m^{(1)} + v_m^{(1)}$ la suma de las variables duales obtenidas al reducir la matriz C.

2. Si hay nodos sin sucesores con etiqueta *No terminal*, procedemos al paso 3. En otro caso, X es la matriz asociada al mejor tour encontrado y z_0 es el costo del tour. FIN.

3. Seleccionamos el nodo $\#p$ sin sucesores, con etiquetado *No terminal* y con la mejor cota w_p .

4. Si las variables de ramificación x's con valor 1 forman un circuito no hamiltoniano, etiquetamos al nodo como *No factible*.

5. Si $w_p \leq z_0$ y los arcos asociados a la matriz X_p forman un circuito hamiltoniano, entonces etiquete al nodo como *Candidato*. Si $w_p < z_0$, entonces actualizamos a z_0 por w_p y a X por X_p.

6. Si $w_p > z_0$, entonces etiquetamos al nodo como *No prometedor*.

7. En otro caso, buscamos un arco (i,j) tal que $\theta(i,j) = \max\theta$ a partir de la matriz $\overline{C}^{(p)}$. Ramificamos a los nodos $\#q$ y $\#(q+1)$ como sucesores de $\#p$, donde $x_{i,j} = 0$ para el nodo $\#q$ y $x_{i,j} = 1$ para el nodo $\#q+1$. Definimos a $\overline{C}^{(q)}, \overline{C}^{(q+1)}$ como copias de $\overline{C}^{(p)}$. Hacemos $\overline{c}^{(q)}_{i,j}. = \infty$, $\overline{c}^{(q+1)}_{i,j} = 0$, $\overline{c}^{(q+1)}_{i,l} = \infty$ y $\overline{c}^{(q+1)}_{k,j} = \infty$, para $k,l \in \{1,\ldots,n\}$, $k \neq j$ y $l \neq i$. Aplicamos el algoritmo húngaro a $\overline{C}^{(q)}, \overline{C}^{(q+1)}$ para obtener a las matrices $X^{(q)}, X^{(q+1)}$ y definimos $w_q = w_p + \sum_{m=1}^{n}(u_m^{(q)} + v_m^{(q)})$ y $w_{q+1} = w_p + \sum_{m=1}^{n}(u_m^{(q+1)} + v_m^{(q+1)})$. Redefinimos $q = q+2$ e ir al paso 2.

Observación 2.7. Sean C una matriz de costos y (u,v) la solución dual óptima obtenida al resolver el problema de asignación asociado a C. Notemos que podemos definir una matriz de costos reducidos $\overline{C}$ a partir de C y (u,v). Suponiendo que a $\overline{C}$ se le hace alguna de las dos modificaciones del algoritmo anterior, entonces el vector (u,v) define una solución dual factible. Por la argumentación anterior y por los teoremas de dualidad, podemos asegurar que las cotas w's son el valor óptimo de resolver el problema de asignación asociado a las matrices $\overline{C}$'s modificadas.

Proposición 2.8. *El algoritmo de ramificación y acotamiento anterior resuelve el PAV.*

Demostración. Sea $R = (X, A, c)$ una red dirigída, completa y acíclica, donde $X = \{1,\ldots,n\}$ y $c : A \to [0,\infty)$. P.d. El algoritmo se ejecuta en tiempo finito.

Vemos que el árbol formado por la ramificación de una instancia está acotado estrictamente por el caso donde se ven todas las posibles combinaciones para las variables $x_{i,j}$, para $i,j \in \{1,\ldots,n\}$, $i \neq j$. Notemos que el problema PAV tiene $n(n-1)$ variables. Al ver todas las posibles combinaciones, se puede construir un árbol con $2^{n(n-1)}$ nodos, pues $2^{n(n-1)} = \sum_{k=1}^{n(n-1)} 2^{k-1} < \infty$ y el k-ésimo nivel del árbol tendría 2^{k-1} nodos. Por lo tanto, el algoritmo se ejecuta en tiempo finito.

P.d. El algoritmo encuentra el valor óptimo.

Sea S el conjunto de circuitos hamiltonianos que se pueden formar en la red. Como el algoritmo se ejecuta en tiempo finito, podemos asegurar que en algún momento todos los nodos con la etiqueta *No terminal* tendrán sucesores. Cuando esto suceda, tenemos los siguientes casos:

- Los nodos con el etiquetado *No factible* no contienen soluciones factibles;
- los nodos con el etiquetado *Candidato* tienen como solución óptima a un tour de costo mínimo, el cual contiene a un subconjunto de tours $S' \subseteq S$;
- es posible que los nodos con el etiquetado *No prometedor* contengan un conjunto $S'' \subseteq S$.

Podemos asegurar que la unión de las regiones de búsqueda de los nodos con etiquetado *Candidato* y *No prometedor* contiene a S. Además, el tour más barato de los nodos candidatos es más barato que cualquiera de los no prometedores por construcción. Por lo tanto, el algoritmo resuelve el PAV.

□

Para este punto, tenemos los conocimientos necesarios para encontrar el mejor tour del ejemplo 2.1. Empecemos definiendo a $z_0 = \infty$. La información de la tabla 2.1 puede ser plasmada en la siguiente matriz de costos.

$$C = \begin{pmatrix} \infty & 64 & 83 & 107 & 188 & 155 \\ 64 & \infty & 80 & 86 & 159 & 120 \\ 83 & 80 & \infty & 111 & 152 & 135 \\ 107 & 86 & 111 & \infty & 114 & 97 \\ 188 & 159 & 152 & 114 & \infty & 141 \\ 155 & 120 & 135 & 97 & 141 & \infty \end{pmatrix}. \tag{2.5}$$

Al aplicar el algoritmo húngaro en la matriz anterior, obtenemos nuestra primer matriz reducida $\overline{C}^{(1)}$, la cual es perteneciente al primer nodo del árbol de ramificación y acotamiento.

$$\overline{C}^{(1)} = \begin{pmatrix} \infty & 0 & 0 & 56 & 93 & 77 \\ 0 & \infty & 0 & 38 & 67 & 45 \\ 0 & 0 & \infty & 44 & 41 & 41 \\ 21 & 3 & 9 & \infty & 0 & 0 \\ 58 & 32 & 6 & 0 & \infty & 0 \\ 42 & 10 & 6 & 0 & 0 & \infty \end{pmatrix}.$$

La asignación del algoritmo húngaro fueron los arcos $(1,3),(3,2),(2,1),(4,5),(5,6)$ y $(6,4)$, la cual es una solución con subtours y por lo tanto, no factible. El valor de esta asignación es de 579, la cual define a la cota $w_1 = 579$. Observemos que hay dos ceros por cada fila y por columna, entonces $\max\theta = 0$, por lo que podemos tomar a $(1,2)$ para ramificar. Definimos a $\overline{C}^{(2)}$ y $\overline{C}^{(3)}$ como copias de $\overline{C}^{(1)}$. Además, hacemos $\overline{c}^{(2)}_{1,2} = \infty$, $\overline{c}^{(3)}_{2,1} = \infty$, $\overline{c}^{(3)}_{1,l} = \infty$ y $\overline{c}^{(3)}_{k,2} = \infty$, con $l \in \{2,3,4,5,6\}$ y $k \in \{1,3,4,5,6\}$. Al reducir las matrices, finalmente obtenemos las siguientes matrices

$$\overline{C}^{(2)} = \begin{pmatrix} \infty & \infty & 0 & 56 & 93 & 77 \\ 0 & \infty & 0 & 38 & 67 & 45 \\ 0 & 0 & \infty & 44 & 41 & 41 \\ 21 & 3 & 9 & \infty & 0 & 0 \\ 58 & 32 & 6 & 0 & \infty & 0 \\ 42 & 10 & 6 & 0 & 0 & \infty \end{pmatrix}, \quad \overline{C}^{(3)} = \begin{pmatrix} \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 38 & 67 & 45 \\ 0 & \infty & \infty & 44 & 41 & 41 \\ 21 & \infty & 9 & \infty & 0 & 0 \\ 58 & \infty & 6 & 0 & \infty & 0 \\ 42 & \infty & 6 & 0 & 0 & \infty \end{pmatrix}.$$

Al aplicar el algoritmo húngaro en las matrices anteriores, no obtenemos cambios en las matrices. Como no se realizó alguna actualización dual durante la reducción de las matrices anteriores, las cotas son las mismas que las del nodo #1. Hacemos $w_2 = 579$ y $w_3 = 579$. La asignación del nodo #1 es la misma que la del nodo #3. Los

arcos asignados en el nodo #2 son $(1,3),(3,2),(2,1),(2,1),(4,5),(5,6)$ y $(6,4)$. Procedemos a ramificar sobre el nodo #2. El único arco con penalización distinta de cero es $(1,3)$, con $\theta(1,3)=56$, pues los demás ceros comparten fila o columna con otro cero, por lo que ramificamos sobre la variable $x_{1,3}$. Definimos a $\overline{C}^{(4)}$ y $\overline{C}^{(5)}$ como copias de $\overline{C}^{(2)}$. Hacemos $\overline{c}^{(4)}_{1,3}=\infty$, $\overline{c}^{(5)}_{3,1}=\infty$, $\overline{c}^{(5)}_{1,l}=\infty$ y $\overline{c}^{(5)}_{k,3}=\infty$, con $l\in\{2,3,4,5,6\}$ y $k\in\{1,2,4,5,6\}$. Al reducir las matrices, obtenemos lo siguiente

$$\overline{C}^{(4)}=\begin{pmatrix}\infty&\infty&\infty&0&37&21\\0&\infty&0&41&70&48\\0&0&\infty&47&44&44\\18&0&6&\infty&0&0\\55&29&3&0&\infty&0\\39&7&3&0&0&\infty\end{pmatrix},\quad \overline{C}^{(5)}=\begin{pmatrix}\infty&\infty&0&\infty&\infty&\infty\\0&\infty&\infty&38&67&45\\\infty&0&\infty&44&41&41\\21&3&\infty&\infty&0&0\\58&32&\infty&0&\infty&0\\42&10&\infty&0&0&\infty\end{pmatrix}.$$

La suma de las variables duales de la matriz $\overline{C}^{(4)}$ es 59, por lo que $w_4=59+w_2=638$ y los arcos asignadas son $(1,4),(4,2),(2,3),(3,1),(5,6)$ y $(6,5)$. Los arcos asignados en el nodo #5 son los mismos que en el nodo #2, por lo que $w_5=579$. Tenemos que w_3 y w_5 son las mejores cotas hasta el momento. Procedemos a ramificar sobre el nodo #3. Vemos que $\theta(2,3)=38+6=44$ y $\theta(3,1)=21+41=62$, los demás ceros de la matriz tienen penalizaciones iguales a cero o infinitas. Con lo anterior, ramificamos sobre la variable $x_{3,1}$. Definimos a $\overline{C}^{(6)}$ y $\overline{C}^{(7)}$ como copias de $\overline{C}^{(3)}$. Hacemos $\overline{c}^{(6)}_{3,1}=\infty$, $\overline{c}^{(7)}_{1,3}=\infty$, $\overline{c}^{(7)}_{3,l}=\infty$ y $\overline{c}^{(7)}_{k,1}=\infty$, con $l\in\{1,2,4,5,6\}$ y $k\in\{2,3,4,5,6\}$. Al reducir las matrices, tenemos que

$$\overline{C}^{(6)}=\begin{pmatrix}\infty&0&\infty&\infty&\infty&\infty\\\infty&\infty&0&38&67&45\\\infty&\infty&\infty&3&0&0\\0&\infty&9&\infty&0&0\\37&\infty&6&0&\infty&0\\21&\infty&6&0&0&\infty\end{pmatrix},\quad \overline{C}^{(7)}=\begin{pmatrix}\infty&0&\infty&\infty&\infty&\infty\\\infty&\infty&0&38&67&45\\0&\infty&\infty&\infty&\infty&\infty\\\infty&\infty&9&\infty&0&0\\\infty&\infty&6&0&\infty&0\\\infty&\infty&6&0&0&\infty\end{pmatrix}.$$

La suma de las variables duales de $\overline{C}^{(6)}$ es 62, por lo que $w_7=62+w_3=641$ y el tour asignado es $\{(1,2),(2,3),(3,5),(5,6),(6,4),(4,1)\}$. Actualizamos a $z_0=641$ y etiquetamos al nodo #7 de *candidato*. Los arcos asignados del nodo #6 son los mismos que los del nodo #3, por lo que $w_7=579$. Las mejores cotas hasta el momento son w_5 y w_7. Ramificamos al nodo #5. Notemos que $\theta(2,1)=59$ y $\theta(3,2)=44$, los demás ceros de la matriz tienen penalizaciones iguales a cero o infinitas. Basado en lo anterior, ramificamos sobre la variable $x_{2,1}$. Definimos a $\overline{C}^{(8)}$ y $\overline{C}^{(9)}$ como copias de $\overline{C}^{(5)}$. Hacemos $\overline{c}^{(8)}_{2,1}=\infty$, $\overline{c}^{(9)}_{1,2}=\infty$, $\overline{c}^{(9)}_{2,l}=\infty$ y $\overline{c}^{(9)}_{k,1}=\infty$, con $l\in\{1,3,4,5,6\}$ y $k\in\{2,3,4,5,6\}$. Al reducir las matrices, tenemos que

$$\overline{C}^{(8)}=\begin{pmatrix}\infty&\infty&0&\infty&\infty&\infty\\\infty&\infty&\infty&0&29&7\\\infty&0&\infty&44&41&41\\0&3&\infty&\infty&0&0\\37&32&\infty&0&\infty&0\\21&10&\infty&0&0&\infty\end{pmatrix},\quad \overline{C}^{(9)}=\begin{pmatrix}\infty&\infty&0&\infty&\infty&\infty\\0&\infty&\infty&\infty&\infty&\infty\\\infty&0&\infty&44&41&41\\\infty&3&\infty&\infty&0&0\\\infty&32&\infty&0&\infty&0\\\infty&10&\infty&0&0&\infty\end{pmatrix}.$$

La suma de las variables duales de $\overline{C}^{(8)}$ es 59, por lo que $w_8 = 59 + w_3 = 638$ y los arcos asignadas son $(1,3),(3,2),(2,4),(4,1),(5,6)$ y $(6,5)$. Los arcos asignados del nodo #9 son los mismos que los del nodo #5, por lo que $w_9 = 579$. Las mejores cotas hasta el momento son w_6 y w_9. Procedemos a ramificar sobre el nodo #7. La única penalización finita y positiva es $\theta(2,3) = 38+6 = 44$. Ramificamos sobre la variable $x_{2,3}$. Definimos a $\overline{C}^{(10)}$ y $\overline{C}^{(11)}$ como copias de $\overline{C}^{(7)}$. Hacemos $\overline{c}^{(10)}_{2,3} = \infty$, $\overline{c}^{(11)}_{3,2} = \infty$, $\overline{c}^{(11)}_{2,l} = \infty$ y $\overline{c}^{(11)}_{k,3} = \infty$, con $l \in \{1,3,4,5,6\}$ y $k \in \{1,2,4,5,6\}$. Al reducir las matrices, tenemos que

$$\overline{C}^{(10)} = \begin{pmatrix} \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 29 & 7 \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 0 & 0 \\ \infty & \infty & 0 & 0 & \infty & 0 \\ \infty & \infty & 0 & 0 & 0 & \infty \end{pmatrix}, \quad \overline{C}^{(11)} = \begin{pmatrix} \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & 0 & \infty & 0 \\ \infty & \infty & \infty & 0 & 0 & \infty \end{pmatrix}.$$

La suma de las variables duales de $\overline{C}^{(10)}$ es 44, por lo que $w_{10} = 44 + w_6 = 623$ y el tour asignado es $\{(1,2),(2,4),(4,5),(5,6),(6,3),(3,1)\}$. Actualizamos a $z_0 = 623$ y etiquetamos al nodo #10 de *candidato*. Notemos que las variables sobre las que se ha ramificado al nodo #11 llevan a un circuito no hamiltoniano ($x_{1,2}, x_{2,3}, x_{3,1} = 1$), entonces etiquetamos al nodo #11 de *no factible*. Las mejores cotas hasta el momento son w_9 y w_{11}. Procedemos a ramificar a w_9. La única penalización finita y positiva es $\theta(3,2) = 41+3 = 44$. Ramificamos sobre la variable $x_{3,2}$. Definimos a $\overline{C}^{(12)}$ y $\overline{C}^{(13)}$ como copias de $\overline{C}^{(9)}$. Hacemos $\overline{c}^{(12)}_{3,2} = \infty$, $\overline{c}^{(13)}_{2,3} = \infty$, $\overline{c}^{(13)}_{3,l} = \infty$ y $\overline{c}^{(13)}_{k,2} = \infty$, con $l \in \{1,2,4,5,6\}$ y $k \in \{1,3,4,5,6\}$. Al reducir las matrices, tenemos que

$$\overline{C}^{(12)} = \begin{pmatrix} \infty & \infty & 0 & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 3 & 0 & 0 \\ \infty & 0 & \infty & \infty & 0 & 0 \\ \infty & 29 & \infty & 0 & \infty & 0 \\ \infty & 7 & \infty & 0 & 0 & \infty \end{pmatrix}, \quad \overline{C}^{(13)} = \begin{pmatrix} \infty & \infty & 0 & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & 0 & \infty & 0 \\ \infty & \infty & \infty & 0 & 0 & \infty \end{pmatrix}.$$

Notemos que las variables sobre las que se ha ramificado al nodo #13 llevan a un circuito no hamiltoniano ($x_{3,2}, x_{2,1}, x_{1,3} = 1$), entonces etiquetamos al nodo #13 de *no factible*. La suma de las variables duales de $\overline{C}^{(12)}$ es 44, por lo que $w_{12} = 44 + w_9 = 623$ y el tour asignado es $\{(1,2),(2,4),(4,5),(5,6),(6,3),(3,1)\}$. Las cotas $w_4 = 638$ y $w_8 = 638$ son mayores que nuestra $z_0 = 623$, por lo que etiquetamos a los nodos #4 y #8 como *no prometedores*.

Para este punto, no queda ningún nodo por ramificar. La ruta que deben de realizar los pilotos es Ciudad de México, Guadalajara, Los Cabos, Tijuana, Hermosillo, Monterrey y regresar a Ciudad de México. El tiempo promedio de esta ruta es de 10 horas con 38 minutos, por lo que los pilotos se pueden turnar el avión durante el recorrido para cumplir sus contratos laborales. En la figura 2.9, se encuentra el árbol del procedimiento anterior.

Durante la ramificación, obtuvimos dos tours con el mismo valor, de hecho, son el mismo tour pero uno está recorrido en el sentido opuesto del otro. Esto se debe a que contamos con un problema simétrico. En este caso, podemos modificar al algoritmo tal que al hacer la variable $x_{i,j} = 0$ por medio de modificar el costo a $c_{i,j} = \infty$, también prohibiremos $x_{j,i}$ haciendo $c_{j,i} = \infty$. Esto nos pudo haber ahorrado algunos nodos. Otra observación importante es que estamos trabajando con árboles, y estos crecen de manera exponencial. En caso de resolver instancias grandes del PAV, el algoritmo se vuelve ineficiente. Hablaremos más sobre esto en el capítulo 4.

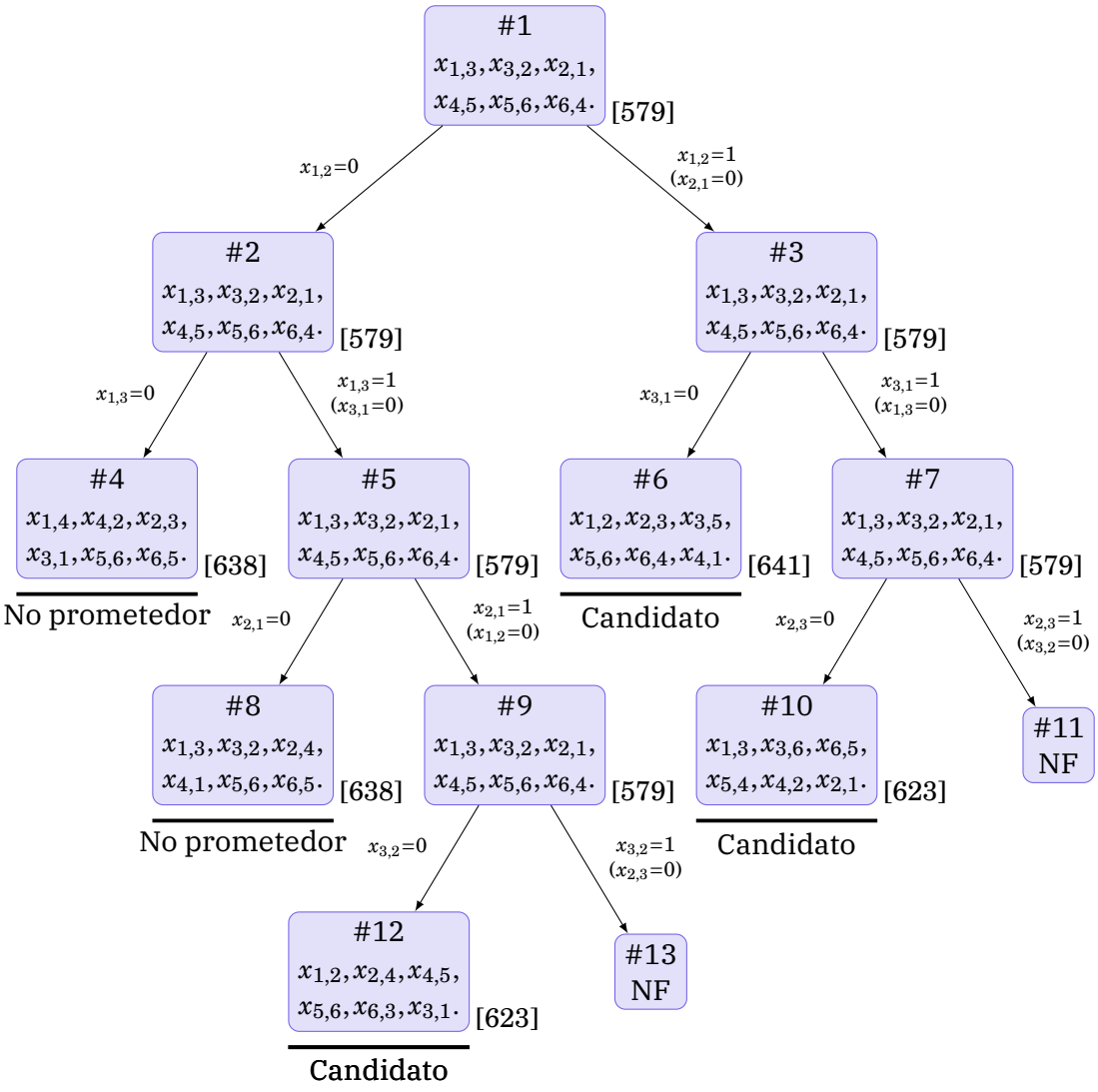


Figura 2.9: El algoritmo de ramificación y acotamiento para el ejemplo 2.1.

Capítulo 3

Ordenamiento de fragmentos

Como ya mencionamos en el primer capítulo, las moléculas de ADN son secuencias de bases nitrogenadas, las cuales son representadas con letras. Conocer dicha secuencia necesita de la obtención de lecturas, las cuales son trozos de la secuencia original. La secuenciación *de novo* consiste en secuenciar una molécula de ADN sin una secuencia de referencia, según Gusfield [1, pág. 159] en el libro *Integer linear programming in computational and systems biology*. Este procedimiento es conocido como secuenciación por escopeta.

En síntesis, la **secuenciación por escopeta** consta de cuatro pasos principales. El primer paso consiste en extraer el ADN a secuenciar para realizar varias copias por medio de una PCR. El segundo paso es fragmentar las secuencias de manera aleatoria. El tercer paso es secuenciar las lecturas. Estos tres primeros pasos ya fueron presentados en el primer capítulo. El último paso consiste tratar los datos y determinar el orden de las lecturas para conocer la secuencia original. Una metodología para ordenar las lecturas es planteando el problema como un problema de mensajero, que posteriormente será reformulado mediante una instancia del PAV. En la figura 3.1 se encuentra ilustrado el proceso anterior.

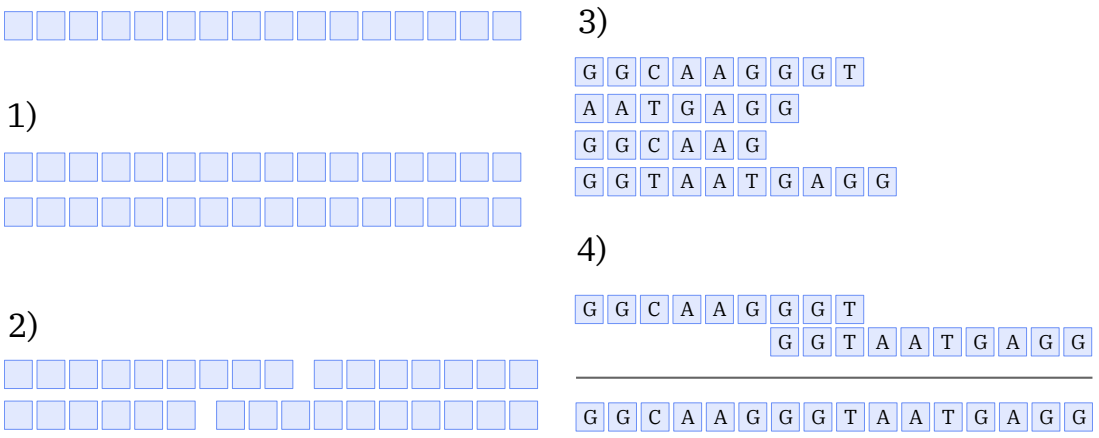


Figura 3.1: Secuenciación por escopeta.

Este capítulo está dividido en tres secciones. En la primera sección veremos los posi-

bles errores que ocurren durante la SBS y cómo interpretar los datos computacionales obtenidos en el procedimiento. En la segunda sección veremos la relación entre el ensamblaje de lecturas y el PAV, construyendo una red dirigida, completa y sin bucles por medio de ellas y sus traslapes. En la tercera sección hablaremos de cómo manejar los errores cometidos durante la secuenciación de lecturas en el problema de optimización y en la obtención de la secuencia.

3.1. Errores e interpretación de datos computacionales

Antes de ver el procesamiento de la información obtenida tras la SBS, necesitamos saber cuáles son las desventajas de esta secuenciación y entender cómo interpretar los datos obtenidos en el procedimiento. En la actualidad, existen muchas compañías que desarrollan tecnología de secuenciación de segunda generación, pero recordemos que en nuestro caso describiremos la tecnología de *Illumina*.

3.1.1. Errores de secuenciación

Las lecturas son secuencias formadas por las letras A, C, T y G obtenidas tras el proceso de SBS. Como ya mencionamos en el capítulo 1, cada una de estas lecturas se obtiene a partir de fotografiar una agrupación de miles de ellas, las cuales crean una señal lumínica para identificar las bases. El problema con las imágenes es que no necesariamente serán 100 % nítidas, ya que durante la secuenciación pueden suceder errores. Nos basaremos en los trabajos de Pfeiffer, Gröber, Blank y col. [18] y de Massingham y Goldman [19] para describir algunos de estos errores de secuenciación en el proceso de la SBS.

Los principales errores de la SBS que mencionan son: diafonía entre agrupamientos adyacentes, errores de fase y atenuación. Recordemos que durante la preparación del ADN a secuenciar, es decir, el primer paso de la figura 3.1, es posible tener errores de mutación. Estos errores son inherentes de la PCR, como explicamos en el capítulo 1. Aún así, las tasas de mutación son bajas y afectan poco al resultado final de la secuenciación.

Una dificultad que se presenta al realizar las fotografías es la identificación de agrupamientos, pues en la imagen sólo se ven puntos de colores. Nos referimos a las **diafonías entre agrupamientos adyacentes** cuando dos agrupamientos no son distinguidos al realizar las fotografías. En lugar de observar dos círculos en la fotografía, se percibe un óvalo. Tras el paso de varios ciclos, es más probable que se pueda diferenciar a los dos agrupamientos. En la figura 3.2 se muestran dos agrupamientos (C).

Los **errores de fase** son aquellos donde una copia en un agrupamiento no va a la par de los ciclos realizados. Los de prefase ocurren cuando la lectura es más corta que la cantidad de ciclos y los de posfase cuando la lectura es más larga. La razón por la que ocurren estos errores es por un mal lavado de las bases nitrogenadas durante cada ciclo de la SBS y por no retirar los terminadores fluorescentes. Este tipo de errores se pueden minimizar con los lavados entre los ciclos. En la figura 3.2 se muestra un ejemplo de ambos errores, donde una copia tiene un error de prefase (C) y una copia

tiene uno de posfase (T).

La **atenuación** es un fenómeno donde no se logra distinguir a los tintes fluorescentes en los terminadores. Esto es debido a que las moléculas son dañadas por la exposición a la luz emitida para poder observarlas. Dicho error tiende a ocurrir cuando la secuenciación lleva varios ciclos. Para evitar este error, la secuenciación de la lectura no debe exceder cierta cantidad de ciclos. En la figura 3.2 se muestra la atenuación en los tintes (T).

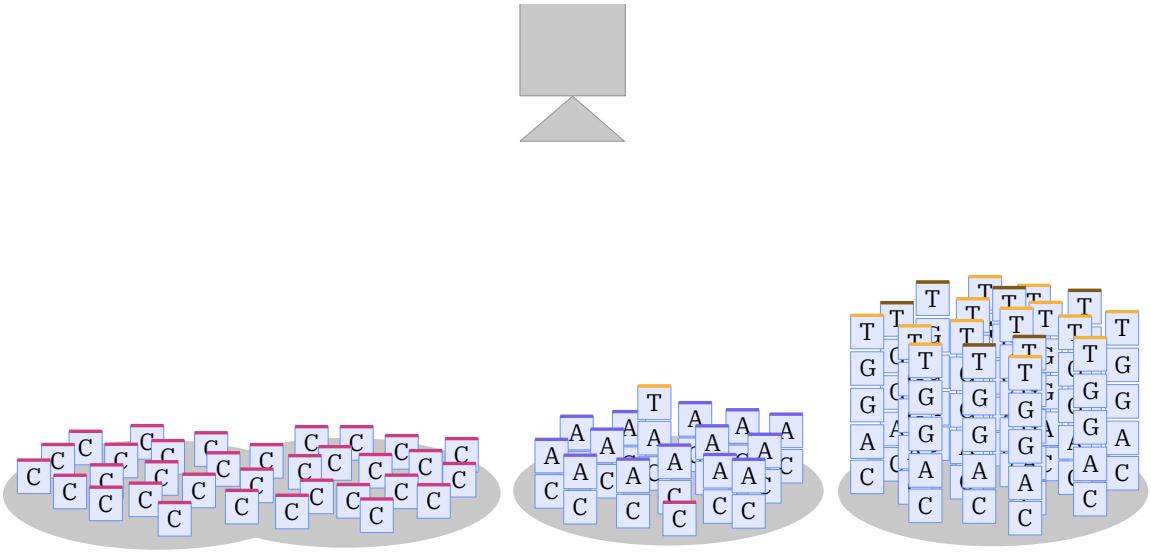


Figura 3.2: Errores de agrupamiento, fase y atenuación.

Ya con las fotografías, el secuenciador realizará un análisis de imágenes. Durante el análisis se utilizan algoritmos para determinar los agrupamientos y la intensidad de las 4 bases en cada uno de ellos. Como resultado, se obtiene un archivo de intensidad de agrupamientos en un formato `.cif`. En este archivo, se almacenan las intensidades de los distintos agrupamientos en la celda. Ya con el archivo de intensidad lumínica y considerando los errores anteriores, el siguiente paso es determinar el orden de bases que conforman cada lectura. Esto es por medio del proceso conocido como **llamada de bases** [20].

Como ya hemos comentado, no se tendrá la certeza de que las bases secuenciadas sean 100 % correctas. Consecuentemente, se introduce una medida de calidad. Dada la fotografía de una base nitrogenada $N \in \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}$ en el proceso de SBS, definimos a Q como la **calidad de la base** N dada por $Q = -10\log_{10}(p)$, donde p es la probabilidad de que la base secuenciada sea incorrecta [21]. Esta probabilidad es estimada por el software del secuenciador durante la SBS para cada base [19]. Notemos que mientras más grande sea Q, menor es la probabilidad de que la base sea incorrecta, esto puede ser observado en la figura 3.3.

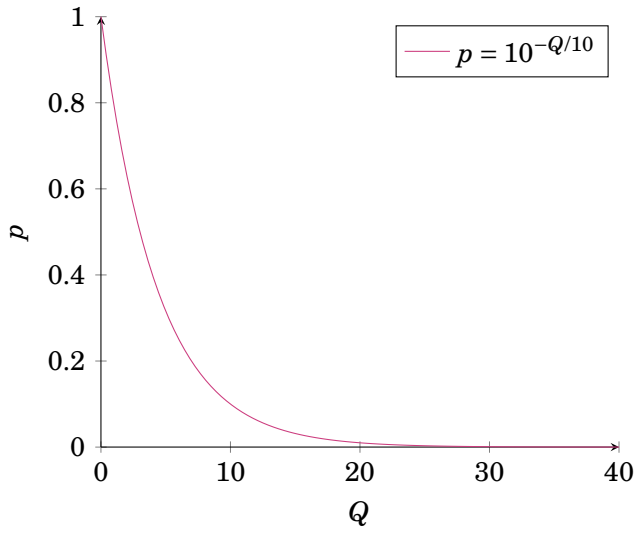


Figura 3.3: Probabilidad de que una base sea incorrecta en función de su calidad.

3.1.2. Formato FASTQ

En el primer capítulo comentamos que tras el procedimiento de SBS, el secuenciador almacena las lecturas en un archivo [3]. Un formato común para plasmar la información de las lecturas es el formato FASTQ, según Cock, Fields, Goto y col. [21]. Antes de manejar las secuencias, es necesario dar una definición matemática para las cadenas y otras definiciones complementarias de ellas [22, pág. 13–14], ya que serán fundamentales durante el resto de este trabajo.

Definición 3.1. Definimos un **alfabeto** Γ como un conjunto finito de cardinalidad mayor o igual a 2. A los elementos de un alfabeto se les llama **letras** o **caracteres**. Definimos **las cadenas (de caracteres)** de Γ de longitud n como el conjunto de funciones $\Gamma_n = \{\omega : \{1,\dots,n\} \to \Gamma\}$, con $n \in \mathbb{N}$. Dichas cadenas suelen ser representadas como una secuencia de n caracteres $\omega = \gamma_1\gamma_2\dots\gamma_n$ donde $\gamma_i \in \Gamma$ para $i \in \{1,\dots,n\}$. Definimos un **lenguaje** L del alfabeto Γ como un conjunto de cadenas y a cada elemento de L se le denomina **palabra**. Definimos al **lenguaje de todas las posibles cadenas de** Γ, $\Gamma^* = \bigcup_{n\in\mathbb{N}} \Gamma_n$, incluida la **cadena vacía** denotada como $\epsilon : \varnothing \to \Gamma$. Finalmente, definimos la **longitud de una cadena** como una función $|\cdot| : \Gamma^* \to \mathbb{N}$ donde $|\omega| = n$ si $\omega \in \Gamma^n$.

Ejemplo 3.1. Los conjuntos $\Gamma_1 = \{0,1\}$ y $\Gamma_2 = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$ son alfabetos, $\omega = \texttt{TTAGGG}$ es una cadena de Γ_2 y $|\omega| = 6$. Un lenguaje de Γ_2 sería $L_1 = \{\texttt{TTAGGG}, \texttt{TAGGCT}, \texttt{AG}\}$, donde una palabra de L_1 es $\texttt{TTAGGG}$. Otro lenguaje de Γ_2 es Γ_2^*, el cual contiene todas las cadenas posibles.

Observación 3.2. a) Las lecturas obtenidas en la SBS son cadenas de caracteres; b) en el contexto de secuenciación, a las subcadenas $\omega \in \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}^*$ de una lectura tales que $|\omega| = k$, se les conoce como k**-meros**, según Lapidus [23, pág. 201].

En los archivos se guardan todas las cadenas correspondientes a lecturas de una

secuenciación. Cada una de ellas es almacenada de la siguiente forma

`@`Título con una descripción opcional.
Lectura expresada con las letras `A`,`C`,`T`,`G` o `N`.
+Repetición del título.
Calidad de la lectura.

La letra `N` está reservada para denotar que no hay información suficiente para identificar la base nitrogenada. La calidad de la lectura es representada por números enteros codificados en un formato ASCII, siglas en inglés de Código Estándar Americano para Intercambio de Información [24, pág. 6]. La calidad Q es un entero entre 0 y 62, el cual es representado por un número del 64 a 126 en el formato ASCII [21], que es un caracter imprimible. Esto es conocido como la codificación PHRED+64. Esta tabla puede ser consultada en [25].

Ejemplo 3.2. Supongamos que contamos con la siguiente lectura perteneciente a un archivo `.fastq`.

```
@J02459.4 Escherichia_phage_Lambda length=12
ATGTGGCGGGTT
+J02459.4 Escherichia_phage_Lambda length=12
EGzo^rq[HeMf
```

Veamos cómo obtener la calidad de las lecturas. La primera lectura tiene asociada una letra `E`. Consultando la tabla ASCII, vemos que dicha letra tiene asociado al número 69. Como se utiliza la codificación PHRED+64, la calidad está trasladada 65 lugares (el conteo empieza en 0), por lo que concluimos que es 4. Al hacer el procedimiento anterior con las demás lecturas, vemos que las calidades son 6, 57, 46, 29, 49, 48, 26, 7, 36, 12 y 37.

3.2. Adaptación del PAV al ordenamiento de lecturas

Una forma de inferir la secuencia del genoma a partir de las lecturas, es resolviendo un problema conocido como el **Problema de la Supercadena Común más Corta** (o **SCS** por sus siglas en inglés) [26, pág. 187], el cual consiste en obtener la cadena más corta que contenga a las lecturas. La ventaja es que podemos resolver el SCS formulándolo como un PM sobre una red específica. En el capítulo 2 vimos cómo resolver al PM a partir del PAV, por lo que nuestro problema se reduce a encontrar el ciclo hamiltoniano más barato. La cuestión es cómo definir a la función de costos.

3.2.1. Prefijos, sufijos y traslapes

Los traslapes entre las cadenas representan la evidencia que tenemos para afirmar que dos cadenas son sucesivas en el ordenamiento. Mientras más grande es el traslape, tenemos una mayor evidencia de que dos cadenas son sucesivas, por esta razón queremos obtener la supercadena común más corta. Para poder resolver el SCS a partir del PM, necesitamos conocer una definición para los traslapes y esta la obtendremos a partir de los prefijos y sufijos de cadenas.

Definición 3.3. Dada una cadena $\omega = \gamma_1\gamma_2\ldots\gamma_n$ de un alfabeto Γ, le llamamos **prefijo** a cualquier subcadena $p_\omega = \gamma_1\gamma_2\ldots\gamma_i$ de ω y le llamamos **sufijo** a cualquier subcadena $s_\omega = \gamma_i\gamma_{i+1}\ldots\gamma_n$ de ω, para $i \in \{1,\ldots n\}$.

Ejemplo 3.3. Sea $\omega = \texttt{GTTTAAGGCGTTT} \in \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}^*$, un prefijo de ω es la cadena $p_\omega = \texttt{GTTTA}$ y un sufijo sería la cadena $s_\omega = \texttt{CGTTT}$.

A partir de los prefijos y sufijos podremos definir a los traslapes. Si un sufijo u resulta ser prefijo de v, entonces tendríamos un traslape entre ambas. Notemos que puede existir más de un traslape entre dos cadenas de un alfabeto Γ, por lo que debe definirse como una relación entre dos cadenas.

Definición 3.4. Dado un alfabeto Γ, definimos al operador de **concatenación de cadenas** como $+ : \Gamma^* \times \Gamma^* \to \Gamma^*$ dado por $u + v = \gamma_1\ldots\gamma_n\gamma_{n+1}\ldots\gamma_m$ donde $u = \gamma_1\ldots\gamma_n$ y $v = \gamma_{n+1}\ldots\gamma_m$. Usualmente el operador se omite y a la concatenación de dos cadenas se le denota con uv. Definimos la **relación de traslapes** de un lenguaje $L \subseteq \Gamma^*$ como $\tau \subseteq L \times L$ donde el par ordenado $(u,v) \in \tau$ si existen tres cadenas $\omega, \omega_1, \omega_2 \in \Gamma^*$ tales que $\omega_1\omega = u$ y $\omega\omega_2 = v$, donde a ω se le llama la **cadena de traslape (o traslape)** entre u y v.

En general, la cadena de traslape entre u y v es sufijo de u y prefijo de v, por lo que hacemos énfasis en que el orden es importante. Observemos que por la definición, $\omega = \epsilon$ también es un traslape, por lo que $\tau = L \times L$. Si restringimos la longitud del traslape, no necesariamente se da la igualdad anterior. La metodología que usaremos para ordenar los fragmentos es por medio de los traslapes. Si dos lecturas λ_i y λ_j tienen un traslape grande, es más probable que λ_j sea una cadena sucesora de λ_i. Estas relaciones pueden ser plasmadas por medio de una red $R = (X, A, c)$.

Antes de continuar, es necesario mencionar que aún considerando que la secuenciación no tuvo errores, al conjunto de lecturas se le debe realizar un tratamiento antes de ser utilizado para plantear la red. Una forma rápida de hacerlo es quedarnos con cadenas lo suficientemente largas para permitir un traslape largo y verificar si existen lecturas que sean subcadenas de otras lecturas para eliminarlas, según Gusfield [1, pág. 160]. Una forma eficiente de detectar las subcadenas es con el algoritmo de Boyer y Moore [27].

3.2.2. Red de traslapes

En el resto de esta sección, veremos cómo realizar el ordenamiento de lecturas para obtener la secuencia original suponiendo que las lecturas no contienen errores de secuenciación. Mencionamos que mientras más grande sea el traslape, mayor evidencia tendremos para decir que dos cadenas son sucesivas. Esto genera un problema ya que el PM es un problema donde se busca el camino hamiltoniano de menor costo. A continuación veremos cómo plantear una red cuya función de costos es tal que a menor costo del arco (i,j) mayor es la evidencia de que λ_i y λ_j son lecturas sucesivas.

Dado el conjunto de lecturas tratadas $\Lambda \subseteq \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}^*$ con $\text{card}(\Lambda) = n$, definimos a $X = \{1,\ldots,n\}$ como una indexación del conjunto. Construimos digráfica $G = (X, A)$ sin bucles y completa de la siguiente manera. Cada nodo $i \in X$ representa al índice de alguna lectura en Λ y cada $(i,j) \in A$ es la relación entre algún sufijo de la lectura λ_i y algún

prefijo de λ_j. A la gráfica $G = (X, A)$ se le conoce como una **gráfica de traslapes**. Definir la función de costos $c : A \to \mathbb{R}$ involucra un problema de optimización. Ejemplifiquemos lo anterior antes de definir a los costos de manera general.

Ejemplo 3.4. Supongamos que tenemos las siguientes dos lecturas de las cuales queremos obtener un traslape:

$$\lambda_1 : \texttt{CGGGTCTGTG},$$
$$\lambda_2 : \texttt{TGTGTTCCGGC}.$$

Notemos que las cadenas 1 y 2 tienen varios traslapes, un traslape $\omega = \texttt{TG}$, pues ω es sufijo de la cadena 1 y prefijo de 2. Una propuesta más apegada a nuestra metodología sería $\omega' = \texttt{TGTG}$, el cual podría ser considerado el mejor traslape entre 1 y 2, ya que no hay un traslape más grande. Un costo que proponemos es $c(1,2) = 10 + 11 - 2 \cdot 4 = 13$. En la figura 3.4 se ilustra la obtención de dicho costo.

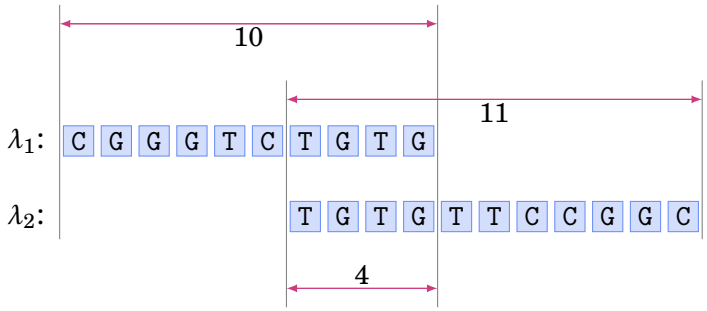


Figura 3.4: Costo de las cadenas del ejemplo 3.4.

En general, la función de costo queda definida como

$$c(i,j) = |\lambda_i| + |\lambda_j| - 2\,\text{máx}\left\{|\omega| : \omega \in \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}^* \text{ es traslape de } (\lambda_i, \lambda_j)\right\}, \tag{3.1}$$

donde $\lambda_i, \lambda_j \in \Lambda$ son las lecturas asociadas a los índices $i, j \in X$ respectivamente. Notemos que la función anterior es no negativa, ya que dado un traslape ω de dos cadenas λ_i y λ_j, se satisface que $|\omega| \leq |\lambda_i|$ y $|\omega| \leq |\lambda_j|$, por lo que $|\lambda_i| + |\lambda_j| - 2|\omega| \geq 0$, para cualquier $\lambda_i, \lambda_j \in \Lambda$. Notemos que de no haber retirado a las lecturas que fueran subcadenas de otras lecturas, tendríamos problemas en definir el costo y existe la posibilidad de que no se pueda definir dicho *traslape* con prefijos y sufijos. Ya con la red $R = (X, A, c)$, podemos resolver el PAV y obtener un ordenamiento de las lecturas para conocer la secuencia original.

Aquí remarcamos que tratar al conjunto de lecturas Λ es importante, ya que de no hacerlo, podríamos tener dificultades con los traslapes y la función de costo. En caso de no quitar las lecturas que son k-meros de otras lecturas, tendríamos problemas pues el traslape se encontraría adentro de otra. Si las lecturas son cortas, no tendremos suficiente información para determinar su posición en el genoma. Los genomas no son aleatorios y siguen patrones dentro de su estructura, razón por la que una lectura corta no es útil.

Comentábamos anteriormente que obtener el tercer término implica resolver un problema de optimización. Para ello, se necesita de un algoritmo de detección de traslapes y seleccionar el máximo de ellos. Dado un alfabeto Γ, podemos encontrar el traslape máximo del par $(u,v) \in \Gamma^* \times \Gamma^*$. Al iterar sobre los posibles sufijos de u, podemos ver si estos son prefijos de v.

Algoritmo de detección de traslapes

1. Dado el par ordenado de cadenas $(u,v) \in \Gamma^* \times \Gamma^*$, definimos a $\ell := \text{mín}\{|u|,|v|\}$ y a $q = 0$.

2. Para $i \in \{1,\ldots,\ell\}$, tomamos al sufijo de u de longitud i, llamado ω. Si ω resulta ser prefijo de v, redefinimos a $q := i$.

3. q es la longitud del traslape máximo del par ordenado (u,v). FIN.

Ejemplo 3.5. Veamos que efectivamente $\omega =$ `TGTG` es la cadena máxima del par (u,v) del ejemplo 3.4. Recordemos que $u =$ `CGGGTCTGTG` y $v =$ `TGTGTTCCGGC`, entonces $\ell = 10 = \text{mín}\{10,11\}$. Las δ's en el cuadro 3.1 indican si existe un traslape en caso de ser iguales a 1.

Tabla 3.1: Detección de traslape.

Paso	ω	δ	q
1	G	0	0
2	TG	1	2
3	GTG	0	2
4	TGTG	1	4
5	CTGTG	0	4
6	TCTGTG	0	4
7	GTCTGTG	0	4
8	GGTCTGTG	0	4
9	GGGTCTGTG	0	4
10	CGGGTCTGTG	0	4

De lo anterior podemos concluir que el traslape máximo tiene una longitud de 4, es decir $\omega =$ `TGTG`.

Ya con un algoritmo de detección de traslapes máximos, podemos plantear una red a partir de un conjunto de lecturas Λ. Dada la red de traslapes y resolviendo en ella el problema del mensajero, se resuelve un problema conocido como **el problema de apareamiento de todos los sufijos y prefijos** (PATSP), el cual es una forma de obtener el orden original de una secuencia de ADN. Esta metodología tiene desventajas pero eso será discutido en la conclusión.

Ejemplo 3.6. Consideremos la secuencia de ADN λ dada por

$$\lambda = \texttt{GAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA}.$$

Queremos obtener dicha secuencia a partir de un conjunto de lecturas $\Lambda = \{\lambda_i\}_{i=1}^{4}$ donde

$$\begin{aligned}\lambda_1 &= \texttt{AAATGGACGCCGGATGACCCCTC},\\ \lambda_2 &= \texttt{GAGACGCCGTTTGGCCTCAAATGG},\\ \lambda_3 &= \texttt{TGTTTTATCTCTGCGA},\\ \lambda_4 &= \texttt{CCCTCCAGCGTGTTT}.\end{aligned}$$

De lo anterior, podemos plantear una red de traslapes, la cual es una red dirigida completa y sin bucles $R = (X, A, c)$. Los costos de dicha red se pueden obtener a partir del algoritmo de detección de traslapes. En dicha red R, al resolver el PM para obtener la solución al SCS. La red resultante es la presentada en la figura 3.5.

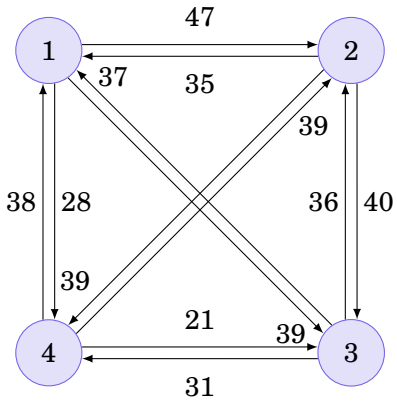


Figura 3.5: Red de traslapes.

En el capítulo 2 vimos que haciendo una modificación a una red donde se necesita resolver un PM, podemos plantear un problema del agente viajero, esto es agregando un nodo, aristas y extendiendo la función de costos. Así, obtenemos la matriz de costos C dada por

$$C = \begin{pmatrix} \infty & 0 & 0 & 0 & 0 \\ 0 & \infty & 47 & 39 & 28 \\ 0 & 35 & \infty & 40 & 39 \\ 0 & 37 & 36 & \infty & 31 \\ 0 & 38 & 39 & 21 & \infty \end{pmatrix}.$$

Ya con la matriz C, podemos aplicar el algoritmo de ramificación y acotamiento visto al final del capítulo 2. El tour obtenido es $t = \{(0,2),(2,1),(1,4),(4,3),(3,0)\}$. Recordemos que debemos quitar los arcos que contengan al nodo cero para obtener la solución del PM en la red R. Podemos construir la cadena original de dos formas: 1) calculando los traslapes nuevamente o 2) utilizando la matriz de costos y las longitudes de las cadenas en Λ, ya que $|\omega_{i,j}| = (c_{i,j} - |\lambda_i| - |\lambda_j|)/2$. Haciendo cualquiera de estos dos procedimientos, obtenemos la cadena original λ.

$$\begin{aligned}\lambda_3 &= \texttt{GAGACGCCGTTTGGCCTCAAATGG}\\ \lambda_4 &= \texttt{AAATGGACGCCGGATGACCCCTC}\\ \lambda_1 &= \texttt{CCCTCCAGCGTGTTT}\\ \lambda_2 &= \texttt{TGTTTTATCTCTGCGA}\\ \lambda &= \texttt{GAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA}\end{aligned}$$

Ya conociendo la secuencia, podemos definir a las coberturas. Dada una cadena $\lambda \in \{\texttt{A},\texttt{C},\texttt{T},\texttt{G}\}^*$ con $|\lambda| = m$ y $\{\lambda_i\}_{i=1}^n$ un conjunto de lecturas de λ, decimos que la base (en la posición i de λ) γ_i tiene una **cobertura** (denotado por $\text{Cob}(\gamma_i)$) igual a k si ya con las lecturas ordenadas, la base aparece en k lecturas diferentes, para $i \in \{1,\dots,n\}$. También se define la **cobertura general** como $\text{CobG}(\lambda) = \left(\sum_{i=1}^n |\lambda_i|\right)/m$.

Ejemplo 3.7. Dado el ejemplo 3.6, la cobertura de cada base nitrogenada sería la siguiente:

```
λ2 =  GAGACGCCGTTTGGCCTCAAATGG
λ1 =                    AAATGGACGCCGGATGACCCCTC
λ4 =                                 CCCTCCAGCGTGTTT
λ3 =                                              TGTTTTATCTCTGCGA
λ  =  GAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA
Cob:  11111111111111111222222111111111111222221111122222111111111111
```

donde la cobertura general es $\text{CobG}(\lambda) = 78/62 \approx 1.258$.

Según Simpson [28], la cobertura juega un papel fundamental para la secuenciación. Para conocer la secuencia del genoma, se necesita tener suficiente evidencia para saber si el ordenamiento es correcto. A mayor sea la cobertura, más información se tiene para realizar la secuenciación. En general, se considera que una cobertura promedio de 30 a 50 es suficiente para realizar el procedimiento anterior. Notemos que el ejemplo 3.6 no satisface estas condiciones, ya que sólo nos sirvió para ilustrar el procedimiento de ordenamiento de lecturas.

Una modificación que se puede realizar a la función de costos es pedir que el traslape entre dos lecturas sea al menos de cierta longitud, de lo contrario hacer ese costo infinito. Esto reduce el espacio de búsqueda de los ciclos hamiltonianos, sin embargo puede suceder que todos los tours de costo finito queden eliminados con esta metodología. Como ejemplo de esto, si hubiéramos pedido un traslape de 6 bases como longitud mínima en el ejemplo 3.6, todos los caminos hamiltonianos tendrían costo infinito y por lo tanto no habría soluciones.

3.3. Ensamblaje con errores de secuenciación

En la sección anterior, consideramos que la secuenciación fue realizada sin errores. Este supuesto es técnicamente improbable que suceda en la práctica, por lo que necesitamos saber cómo manejar los datos sin dicho supuesto. Veamos el siguiente ejemplo.

Ejemplo 3.8. Supongamos que tenemos dos lecturas λ_1 y λ_2. Queremos determinar si hay un traslape largo para el par (λ_1, λ_2).

```
λ1 =  GAGACGCCGTTTGGCCTCCCGCCGGA
                 -- --------- -------
λ2 =             CC_TTTGGCCTCACGCCGGATGAC
```

Observemos que el único traslape posible es ϵ, pero el *traslape* mostrado tiene dos base de diferencia. Esto nos motiva a tratar de corregir los errores de secuenciación, dar una noción de *traslapes aproximados* y ver cómo podemos aplicar esto al ordenamiento. En esta sección, mencionaremos algunos programas de tratamiento y corrección de datos, una métrica de errores y cómo incluirla en la red de traslapes.

3.3.1. Manejo de errores

Tras obtener el archivo `.fastq` producido por la MiSeq de Illumina, es necesario inspeccionar la calidad de los datos y realizarles un tratamiento para posibilitar el planteamiento de una red de traslapes aproximados y así obtener una buena secuenciación. A lo largo de las últimas décadas se han desarrollado varios programas para facilitar las labores anteriores. A continuación, veremos algunos de ellos.

FastQC [29] es un programa desarrollado por el grupo de bioinformática del Instituto de Babraham. Dicho software recibe un archivo `.fastq` con datos crudos a partir del cual se genera un resumen estadístico de la calidad de las lecturas. Dicho resumen incluye la longitud de la cadena, el porcentaje del contenido CG (el cual tiende a ser superior al 50 %) y algunos diagramas de cajas para cada posición de las bases en las lecturas. Este programa también puede ser utilizado para verificar que el conjunto de lecturas sea de buena calidad tras una limpieza de datos.

En este punto, se puede hacer un recorte del conjunto de lecturas para mejorar la calidad o usar algún software de corrección de errores. Hacemos nuevamente énfasis en que la cobertura juega un papel fundamenta, pues a mayor cantidad datos, los algoritmos de corrección suelen tener un mejor desempeño. En breve describiremos la rutina principal de dos algoritmos de detección y corrección de errores.

SHREC es un algoritmo publicado por Schröder, Schröder, Pgulisi y col. [30], el cual se basa en plantear un trie de sufijos a partir del conjunto de lecturas Λ, donde dependiendo de la frecuencia de los sufijos, irá corrigiendo los errores. Dadas las lecturas Λ, un **trie de sufijos** es una estructura de árbol que contendrá a todos los sufijos de las lecturas en Λ. En el ejemplo 3.9 veremos cómo construirlo a partir de un conjunto de lecturas.

Ejemplo 3.9. Dado el conjunto de $\Lambda_1 = \{\texttt{TCTGTG}, \texttt{TATGTTC}, \texttt{GTCTGT}\}$, veamos cómo construir el trie de sufijos. Primero plasmemos a todos ellos en una tabla.

Tabla 3.2: Sufijos de las lecturas del ejemplo 3.9

Longitud	TCTGTGT	TATGTTC	GTCTGT
1	T	C	T
2	GT	TC	GT
3	TGT	TTC	TGT
4	GTGT	GTTC	CTGT
5	TGTGT	TGTTC	TCTGT
6	CTGTGT	ATGTTC	GTCTGT
7	TCTGTGT	TATGTTC	–

Ya con ellos, podemos construir una red dirigida donde cada nodo está asociado a las bases presentes en un sufijo representado por un camino desde la raíz. El nodo inicial indica una cadena vacía de caracteres. En la figura 3.6 se presenta el trie del ejemplo. Podemos ver que 4 sufijos de la tabla 3.2 inician con `TG`, por lo que la `G` debajo

de la primera T en aparecer en la gráfica 3.6 está etiquetada con dicho número.

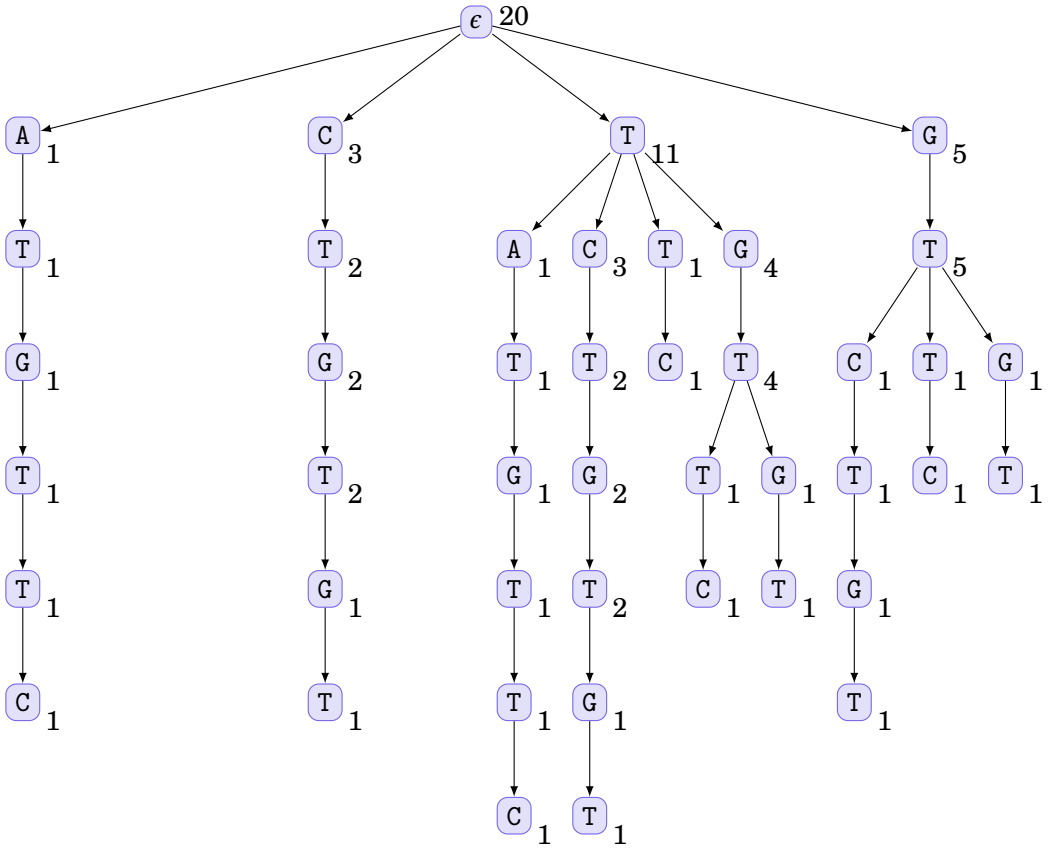


Figura 3.6: Trie del ejemplo 3.9.

Necesitamos obtener un nivel $t \in \mathbb{N}$ en función de Λ para iniciar la detección de errores. Si $t = 2$, podremos ver que las ramas que contienen a los prefijos AT y TA tienen un peso de 1 y una longitud de al menos 6 bases. Observando que el conteo de valor bajo empieza desde A y dicha base no se presenta en otras ramas, podemos inferir que ahí existe un error. Notemos que los sufijo TGTGT y GTGT son casi idénticos a TATGT y ATGT, por lo que G es una letra candidata para corregir dicho error. Finalmente, la lectura corregida sería TGTGTTC.

DecGPU es un algoritmo desarrollado por Liu, Schmidt y Maskell [31], cuya subrutina principal detecta k-meros frecuentes los cuales utiliza para retocar los errores. Las cadenas son encontradas y las correcciones realizadas al resolver un problema conocido como el **problema de alineación espectral**. Veamos el ejemplo 3.10 donde se ilustra una metodología similar.

Ejemplo 3.10. Consideremos al conjunto de lecturas Λ que contiene a

$$\lambda_1 = \texttt{TAGGGTTAGGGTTA},$$
$$\lambda_2 = \texttt{GGTTAGAGTTAGGGT},$$
$$\lambda_3 = \texttt{TTAGGGTTAGGGTTA}.$$

Un 6-mero que se repite 4 veces es TTAGGG. Dicho 6-mero difiere en una base del 6-mero TTAGAG, el cual sólo aparece una vez. Considerando esto, es posible que exista un error en la quinta base A de la subcadena anterior. De ahí se concluye que la lectura λ_2 = GGTTAGGGTTAGGGT.

3.3.2. Una métrica de errores

Lo anterior es útil para reducir el número de errores de secuenciación, pero aún así, no estamos exentos de corregirlos totalmente. Dicho eso, la idea de plantear una red de traslapes carece de sentido. Como ya mencionamos anteriormente, en lugar de pensar en traslapes, nos será útil manejar la idea de *traslapes aproximados*. Para ello, es necesario tener una forma de medir qué tan parecida es una cadena de otra.

Comentamos que en el ejemplo 3.8, necesitábamos realizar dos cambios al sufijo CCGTTTGGCCTCCCGCCGGA de λ_1 para que fuera prefijo de λ_2. Dichos cambios era la eliminación de una G y el intercambio de una C por una A. Otra forma de ver el traslape sería realizar la inserción de una G y el intercambio de una A por una C en el prefijo CCTTTGGCCTCACGCCGGA de λ_2 para que sea sufijo de λ_1. Veamos la definición de estas tres modificaciones.

Definición 3.5. Dado Γ un alfabeto, decimos a las funciones $\delta_E : \Gamma^* \times \mathbb{N} \times \Gamma \to \Gamma^*$, $\delta_I : \Gamma^* \times \mathbb{N} \times \Gamma \to \Gamma^*$ y $\delta_D : \Gamma^* \times \mathbb{N} \to \Gamma^*$ como funciones de **intercambio**, **inserción** y **eliminación** respectivamente si dados $\omega = \gamma_1 \dots \gamma_n \in \Gamma^*$, $i \in \{1, \dots, n\}$ y $\gamma' \in \Gamma$, sus reglas de correspondencia están dadas por $\delta_E(\omega, i, \gamma') = \gamma_1 \dots \gamma_{i-1}\gamma'\gamma_{i+1} \dots \gamma_n$, $\delta_I(\omega, i, \gamma') = \gamma_1 \dots \gamma_{i-1}\gamma_i\gamma'\gamma_{i+1} \dots \gamma_n$ y $\delta_D(\omega, i) = \gamma_1 \dots \gamma_{i-1}\gamma_{i+1} \dots \gamma_n$.

Ejemplo 3.11. Consideremos a λ_1 = TTTGGCCT y a λ_2 = TTAGGCCTA. Veamos qué cambios podemos efectuar en λ_1 para obtener a λ_2.

$$\begin{aligned} \delta_D(\texttt{TTTGGCCT}, 3) &= \texttt{TTGGCCT}, \\ \delta_I(\texttt{TTGGCCT}, 2, \texttt{A}) &= \texttt{TTAGGCCT}, \\ \delta_I(\texttt{TTAGGCCT}, 8, \texttt{C}) &= \texttt{TTAGGCCTC}, \\ \delta_E(\texttt{TTAGGCCTC}, 9, \texttt{A}) &= \texttt{TTAGGCCTA}. \end{aligned}$$

Vemos que tras 4 operaciones, logramos transformar a λ_1 en λ_2, pero se pudo haber obtenido con sólo un intercambio y una inserción.

Del ejemplo anterior, vimos que hay varias formas de modificar a las cadenas para que resulten ser iguales a otras. La cuestión es que no nos sirve hacer muchos cambios si estos se pueden realizar en menor cantidad. Reducirlos nos puede ayudar a definir una métrica para cadenas. Nuestro objetivo es proponer una métrica para el conjunto Γ^*, la cual nos indicará la cantidad de cambios que necesitamos realizar en una cadena $u \in \Gamma^*$ para obtener a la cadena $v \in \Gamma^*$. A dichas métricas para cadenas se les conoce como **distancias de cadenas** [32, pág. 31]. A continuación, una métrica útil para manejar el problema de los traslapes aproximados [32, pág. 37].

Definición 3.6. Dado un alfabeto Γ, se define a la **distancia de Levenshtein** a la función $d_{\mathscr{L}} : \Gamma^* \times \Gamma^* \to \mathbb{R}$ tal que $d_{\mathscr{L}}(u, v)$ es el mínimo número de inserciones, eliminaciones y sustituciones que se le debe realizar a la cadena u para obtener la cadena v.

La distancia de Levenshtein considera a los errores de fase y mutación que sucedan durante la secuenciación de lecturas. Considerando los traslapes aproximados entre las lecturas, podemos utilizar la métrica anterior para penalizarlos. La cuestión es cómo podemos calcular a la distancia entre dos cadenas $u, v \in \Gamma^*$. Erickson [33] comenta que una manera de calcularlo es considerando una definición recursiva. Dadas las cadenas $u, v \in \Gamma^* \setminus \{\epsilon\}$ tales que $|u| = n$ y $|v| = m$, entonces

$$d_{\mathscr{L}}(u,v) = \text{mín} \begin{cases} d_{\mathscr{L}}(u,v') + 1, \\ d_{\mathscr{L}}(u',v) + 1, \\ d_{\mathscr{L}}(u',v') + \mathbb{1}_{\{\gamma_n \neq \gamma_m\}}(\gamma_n, \eta_m), \end{cases}$$

donde γ_n y η_m son las últimas letras de u y v, $u' = \delta_D(u,n)$ y $v' = \delta_D(v,n)$. Notemos que la definición anterior, tiene tres casos distintos en los cuales se considera lo siguiente:

1. se agregó η_m en v;
2. se eliminó a γ_n de u;
3. se realizó un intercambio si $\gamma_n \neq \eta_m$ o no hay modificaciones si $\gamma_n = \eta_m$.

Es sencillo calcular la distancia de cualquier cadena $u \in \Gamma^*$ a la cadena vacía, ya que se deben de agregar todas las letras que componen a u, por lo que $d_{\mathscr{L}}(u,\epsilon) = |u|$. Por conmutatividad de las métricas, tenemos que $d_{\mathscr{L}}(\epsilon, v) = |v|$, para cualquier $v \in \Gamma^*$. Contando con un caso base y una ecuación recursiva, podríamos proceder por recursión, pero es un proceso muy costoso en un sentido computacional. Erickson [33, pág. 9 cap. 5] propone realizar el cálculo por medio de la programación dinámica.

Recordemos que en programación dinámica, queremos dividir el problema en varios subproblemas para resolverlos y a partir de ellos obtener la solución al problema global. Partiremos del hecho de que $d_{\mathscr{L}}(u,\epsilon) = |u| = d_{\mathscr{L}}(\epsilon,u)$ para cualquier cadena u y a partir de ello realizar los cálculos restantes. Dados $u, v \in \Gamma^*$ tales que $|u| = n$ y $|v| = m$, sea $L \in M_{(n+1)\times(m+1)}(\mathbb{R})$ una matriz en función de (u,v) dada por $l_{i,j} = d_{\mathscr{L}}(u_i, v_j)$, donde u_i, v_j son las subcadenas de u y v conformadas por sus primeras i y j letras respectivamente, para $i \in \{0,\dots,n\}$ y $j \in \{0,\dots,m\}$. Por las propiedades ya mencionadas de la distancia de Levenshtein, tenemos que $l_{i,j} = \text{mín}\{l_{i,j-1} + 1, l_{i-1,j} + 1, l_{i-1,j-1} + \mathbb{1}_{\gamma_i \neq \eta_j}\}$, $l_{i,0} = i$, $l_{0,j} = j$ y $l_{0,0} = 0$, para $i \in \{1,\dots,n\}$ y $j \in \{1,\dots,m\}$. Al completar a la matriz L, obtenemos $d_{\mathscr{L}}(u,v) = l_{n,m}$. Esto resulta más eficiente que el método recursivo ya que ahorra varios cálculos [33, pág. 9 cap. 5].

Ejemplo 3.12. Sea $u =$ `GCTT` y $v =$ `ACTTTG`. Calculemos la distancia entre ellos. Observemos que ya conocemos la primera fila y la primera columna de la matriz L.

$$L = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & & & & & & \\ 2 & & & & & & \\ 3 & & & & & & \\ 4 & & & & & & \end{pmatrix}.$$

A partir de aquí, el resto del cálculo consiste en utilizar la recursividad de la distancia de Levenshtein. Como `A` $\neq$ `G`, tenemos que $d_{\mathscr{L}}(\texttt{A},\texttt{G}) = l_{1,1} = \text{mín}\{1+1, 1+1, 0+1\} = 1$. Como

$\texttt{C} \neq \texttt{G}$, tenemos que $d_{\mathscr{L}}(\texttt{AC},\texttt{G}) = l_{1,2} = \text{mín}\{1+1, 2+1, 1+1\} = 1$. Como $\texttt{A} \neq \texttt{C}$, tenemos que $d_{\mathscr{L}}(\texttt{A},\texttt{GC}) = l_{2,1} = \text{mín}\{2+1, 1+1, 1+1\} = 1$. Como $\texttt{C} = \texttt{C}$, tenemos que $d_{\mathscr{L}}(\texttt{AC},\texttt{GC}) = l_{2,2} = \text{mín}\{2+1, 2+1, 1+0\} = 1$. Continuando con los cálculos, la matriz resultante es la siguiente.

$$L = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 1 & 2 & 3 & 4 & 5 & 5 \\ 2 & 2 & 1 & 2 & 3 & 4 & 5 \\ 3 & 3 & 2 & 1 & 2 & 3 & 4 \\ 4 & 4 & 3 & 2 & 1 & 2 & 3 \end{pmatrix}.$$

Navarro [32, pág. 37] comenta que $0 \leq d_{\mathscr{L}}(u,v) \leq \text{máx}\{|u|,|v|\}$, por lo que podemos decir que las cadenas u, v son parecidas en una proporción $p := d_{\mathscr{L}}(u,v)/\text{máx}\{|u|,|v|\} \in [0,1]$. Esto nos será útil para ver qué tanto se parece un genoma conocido con un genoma obtenido por la secuenciación por escopeta y el PAV. Ahora, suponiendo que $|u| > |v|$, entonces $d_{\mathscr{L}}(u,v) \geq |u| - |v|$, pues a v se le necesitan agregar al menos $|u| - |v|$ letras para ser igual a u. En general $d_{\mathscr{L}}(u,v) \geq ||u| - |v||$, por lo que una cota superior de la proporción p es $||u| - |v||/\text{máx}\{|u|,|v|\}$.

Una constante que nos será útil más adelante es $\kappa > 1$ en función del par (u,v) tal que $\kappa d_{\mathscr{L}}(u,v) \leq |u| + |v|$, para cualesquiera $u, v \in \Gamma^* \setminus \{\epsilon\}$. Notemos que $d_{\mathscr{L}}(u,v) < \text{mín}\{|u|,|v|\} + \text{máx}\{|u|,|v|\} = |u| + |v|$, ya que $d_{\mathscr{L}}(u,v) \leq \text{máx}\{|u|,|v|\}$ y porque la única $u \in \Gamma^*$ tal que $|u| = 0$ es $u = \epsilon$. Vemos que κ está dada por

$$\kappa_{u,v} = \frac{\text{mín}\{|u|,|v|\} + \text{máx}\{|u|,|v|\}}{\text{máx}\{|u|,|v|\}},$$

debido a que $\kappa d_{\mathscr{L}}(u,v) \leq \text{máx}\{|u|,|v|\} + \text{mín}\{|u|,|v|\} = |u| + |v|$.

3.3.3. Red de traslapes aproximados

Utilicemos las ideas anteriores para proponer una nueva función de costos en la red R y así obtener el ordenamiento de las lecturas. Recordemos que un traslape del par (u,v) se da cuando un sufijo de u y un prefijo de v coinciden. Si ahora consideramos que es posibles tener errores, comparamos a cada prefijo y sufijo y penalizaremos en función de la distancia entre ellos. Dado el conjunto de lecturas tratadas Λ tal que $\text{card}(\Lambda) = n$, podemos construir a la digráfica completa y sin bucles $G = (X, A)$, donde $X = \{1 \ldots, n\}$ es una indexación de las lecturas. Definimos a la función $c : A \to \mathbb{R}$ dada por

$$c(i,j) = |\lambda_i| + |\lambda_j| - \text{máx}\left\{|s_i| + |p_j| - \kappa_{s_i,p_j} d_{\mathscr{L}}(s_i,p_j) : s_i \text{ es sufijo de } \lambda_i \text{ y } p_j \text{ es prefijo de } \lambda_j\right\},$$

donde $\lambda_i, \lambda_j \in \Lambda$ son las lecturas asociadas a los índices $i, j \in X$. Al tercer término de la ecuación anterior lo llamaremos como **calidad del mejor traslape (aproximado)** y una forma de calcularlo para el arco (i,j), es iterando sobre todos los sufijos y prefijos de λ_i y λ_j, a la par de calcular la distancia de Levenshtein entre ellos. Como $|s_i| \leq |\lambda_i|$ y $|p_j| \leq |\lambda_j|$, tenemos que $c(i,j)$ es no negativo, para toda $(i,j) \in A$.

Ejemplo 3.13. Consideremos a las lecturas

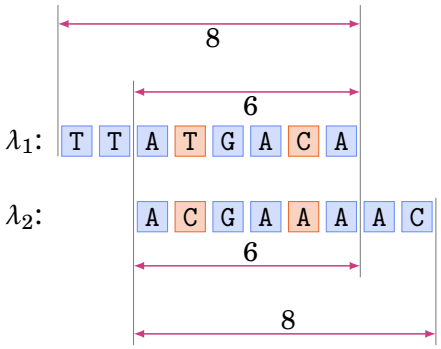
$$\lambda_1 = \texttt{TTATGACA},$$
$$\lambda_2 = \texttt{ACGAAAAC}.$$

Dichas lecturas contienen dos errores cada una, por lo que utilizaremos la definición anterior de la función c para calcular el costo $c(1,2)$. Debemos iterar sobre todos los sufijos de λ_1 y los de prefijos de λ_2 para ver cual es la mejor aproximación del traslape y así obtener el tercer término del costo. Para ello, es necesario encontrar el máximo entre 64 posibles combinaciones. Por lo anterior, en la tabla 3.3 se presenta el cálculo de 10 combinaciones.

Tabla 3.3: Calidad de los traslapes aproximados entre λ_1 y λ_2.

s_1	p_2	$\lvert s_1\rvert$	$\lvert p_2\rvert$	$d_{\mathscr{L}}(s_1,p_2)$	κ_{s_1,p_2}	Calidad del traslape
ACA	ACGAAAA	3	7	4	$1.\overline{428571}$	$4.\overline{285714}$
ATGACA	A	6	1	5	$1.1\overline{6}$	$1.1\overline{6}$
TTATGACA	AC	8	2	6	1.25	2.5
A	ACGA	1	4	3	1.25	1.25
TGACA	ACG	5	3	3	1.6	3.2
TATGACA	ACGAAA	7	6	3	$1.\overline{857142}$	$7.\overline{428571}$
TATGACA	AC	7	2	5	$1.\overline{285714}$	$2.\overline{571428}$
TTATGACA	ACGAAA	8	6	5	1.75	7
GACA	ACGA	4	4	2	2	4
ATGACA	ACGAAA	6	6	2	2	8

Para este ejemplo, el tercer término es el asociado a las cadenas ATGACA y ACGAAA. De lo anterior, tenemos que $c(1,2) = 8+8-8 = 8$, esto es ilustrado en la figura 3.7. La razón por la cual incluimos a las constantes κ's en el costo, es para que reducir la cantidad de sufijos y prefijos que definan al costo, lo cual es útil para la obtención del genoma una vez que se tenga el orden de las lecturas. Sin la constante en el ejemplo anterior, TATGACA y ACGAAA también serían candidatos de un traslape aproximado.



$$c(1,2) = 8+8-\left(6+6-\tfrac{6+6}{6}\cdot 2\right)$$

Figura 3.7: Costo de las lecturas del ejemplo 3.13.

Ejemplo 3.14. Consideremos la secuencia de ADN λ dada por

$$\lambda = \texttt{GAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA}.$$

Queremos obtener dicha secuencia a partir de un conjunto de lecturas $\Lambda = \{\lambda_i\}_{i=1}^{4}$ donde

$$\begin{aligned}
\lambda_1 &= \texttt{GTTTGGCCTCAAATGGACGCAGGATGACCCCT},\\
\lambda_2 &= \texttt{GAGAACGCCGTTTGGCCTCAAATGGACGCCGGA},\\
\lambda_3 &= \texttt{TGACCCCTCCAGCGTGTTTTATCTCTGCGA},\\
\lambda_4 &= \texttt{AGACGCCGGAGACCCCTCCAGCGTGTTT},
\end{aligned}$$

las cuales contienen errores de secuenciación. A partir del conjunto Λ, podemos plantear una red de traslapes aproximados, la cual es una red dirigida completa y sin bucles $R = (X, A, c)$. En dicha red R, podemos resolver el PM para obtener la solución al SCS. La red resultante es la presentada en la figura 3.8.

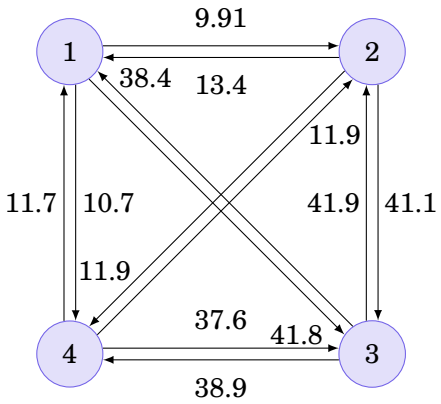


Figura 3.8: Red de traslapes.

En el capítulo 2 vimos que haciendo una modificación a una red donde se necesita resolver un PM, podemos plantear un problema del agente viajero, esto es agregando un nodo, aristas y extendiendo la función de costos. Así, obtenemos la matriz de costos C dada por

$$C = \begin{pmatrix}
\infty & 0 & 0 & 0 & 0\\
0 & \infty & 34.5 & 34.4 & 30.4\\
0 & 19 & \infty & 35.8 & 30.2\\
0 & 38.6 & 36.3 & \infty & 32.8\\
0 & 37.9 & 33.3 & 22 & \infty
\end{pmatrix}.$$

Ya con la matriz C, podemos aplicar el algoritmo de ramificación y acotamiento visto al final del capítulo 2. El tour obtenido es $t = \{(0,2),(2,1),(1,4),(4,3),(3,0)\}$. Recordemos que debemos quitar los arcos que contengan al nodo cero para obtener la solución del PM en la red R. Podemos construir al genoma a partir de los k-meros obtenidos al definir los costos y haciendo un análisis de cadenas aproximadas sobre ellos.

```
λ2 =  GAGAACGCCGTTTGGCCTCAAATGGACGCCGGA
λ1 =          GTTTGGCCTCAAATGGACGCAGGATGACCCCT
λ4 =                      AGACGCCGGA_GACCCCTCCAGCGTGTTT
λ3 =                                 TGACCCCTCCAGCGTGTTTTATCTCTGCGA
λ' =  GAGAACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA
λ  =  GAGA_CGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGA
Cob:  11111111122222222222222333333333323333333222222222211111111111
```

En este caso, la cobertura general de λ' es 123/63, que es aproximadamente 1.9523. Ésta resulta útil al momento de corregir los errores de secuenciación. A mayor cantidad de lecturas, será más fácil corregirlos. Notemos que mientras más frecuente sea una letra en alguna posición (incluyendo los espacios vacíos), es utilizada para ocupar la posición de incertidumbre. El único error que no fue corregido fue la letra `A` en la quinta posición de λ', debido a que no se tienen más lecturas para evidenciar el error. El genoma obtenido con las lecturas y la molécula de referencia coinciden en un 98.41%.

Capítulo 4

Complejidad y heurísticas

En el capítulo 2 se propuso una algoritmo de ramificación y acotamiento para resolver el PAV. Cuando se resuelven instancias pequeñas con dicho algoritmo, no hay problema alguno, ya que se resuelve en poco tiempo. La dificultad surge cuando se necesita resolver instancias grandes, ya que el algoritmo podría llegar a tardar siglos o más tiempo para encontrar un óptimo. Aunado a esto, usualmente las lecturas obtenidas en la secuenciación son muy pequeñas [3, pág. 9] y hay que ordenar una gran cantidad de ellas, por lo que queremos resolver una instancia grande del PAV.

En la actualidad, no se conocen algoritmos que resuelvan eficientemente instancias grandes del PAV. Por el motivo anterior, se opta por el uso de métodos heurísticos, los cuales no garantizan llegar al óptimo pero sí llegan a buenas soluciones de un problema de optimización en poco tiempo. En nuestro caso, nos interesa ver un heurístico que obtenga buenas soluciones para instancias del PAV. En este capítulo, veremos la dificultad computacional del PAV y veremos un método heurístico.

4.1. Teoría de complejidad y el PAV

En síntesis, el PAV consiste en obtener el tour de menor costo de una red completa y sin bucles. A partir de él, podemos definir un nuevo problema al dar un presupuesto k y verificar la existencia de algún tour que cueste a lo más k unidades. El problema anterior es conocido como el problema de decisión asociado al PAV. Los problemas de decisión se caracterizan por ser preguntas cuya respuesta sólo es afirmativa o negativa. A partir de ellos definiremos cuándo un problema es complicado de resolver. En esta sección, daremos una noción matemática de la dificultad de un problema a partir de la teoría de complejidad computacional y demostraremos que el PAV es un problema NP−Duro, los cuales se consideran como *problemas difíciles* de resolver. Para esta sección, utilizaremos principalmente el libro *Introduction to Theory of Computation* de Sipser [22] hasta nuevo aviso.

4.1.1. Máquinas de Turing

Para este punto necesitamos dar una noción formal de lo que conocemos como algoritmo. El modelo utilizado hasta la fecha es el que fue propuesto por Alan Turing en 1936 y se llama *máquina de Turing*. Supongamos que tenemos una cinta de longitud infinita

hacia ambos lados, dividida por celdas y que contendrá las letras de una cadena $\omega \in \Gamma^*$, donde Γ es un alfabeto, y espacios en blanco en el resto de celdas. Se suele reservar el símbolo $\sqcup$ para los espacios vacíos o en blanco de la cinta. La primera cadena se irá modificando a partir de un alfabeto Σ tal que $\sqcup \in \Sigma$ y $\Gamma \subseteq \Sigma$. Ahora, las modificaciones son realizadas por una cabecilla que inicialmente se encuentra sobre la celda de la primera letra de ω. Dicha cabecilla puede cambiar de estado, sobrescribir la celda observada y moverse a la izquierda o derecha dependiendo de su estado y la letra que observe (vea la figura 4.1). La parte esencial para dar formalidad a la noción anterior, es poder conceptualizar la cabecilla y sus movimientos en una función llamada función de transición.

Definición 4.1. Sean Γ, Σ alfabetos tal que $\sqcup \in \Sigma$ y $\Gamma \subseteq \Sigma$ y Q un conjunto finito (llamado **conjunto de estados**), definimos una **función de transición determinista** como una función $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 1\}$; es decir, dados el estado de la cabecilla y la lectura de la celda en la cinta, podemos cambiar el estado de la cabecilla, reescribir sobre la celda y mover la cabecilla a la izquierda o derecha. Definimos a una **maquina de Turing determinista** M como una tupla $M = (Q, \Gamma, \Sigma, \delta)$ donde Q es un conjunto de estados tal que $q_0, q_a, q_r \in Q$ son los estados inicial, de aceptación y de rechazo respectivamente, Γ es un alfabeto y $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 1\}$ es una función de transición.

Mientras que no se llegue a un estado final, la máquina de Turing irá cambiando la localización de la cabecilla, el contenido de la cinta y actualizará su estado. Lo anterior es representado en su **configuración**, la cual se denota con $u\,q\,v$, donde $u, v \in (\Sigma \setminus \{\sqcup\})^*$ son cadenas sin espacios tal que uv es la cadena actual de la cinta, $q \in Q$ es el estado actual de la máquina y la cabecilla se encuentra localizada en la primera letra de v.

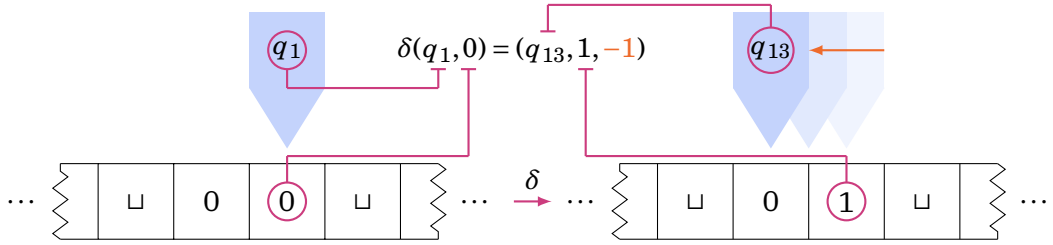


Figura 4.1: Máquina de Turing de configuración $00q_1$ a $0q_{13}1$.

Ejemplo 4.1. Definamos una máquina de Turing que reciba enteros positivos en base binaria (sin ceros a la izquierda) y los duplique. La máquina debe aceptar cadenas del lenguaje L definido por $L = \{\omega \in \{0,1\}^* : \omega = \gamma_1 \ldots \gamma_n,\ \gamma_1 = 1,\ n \in \mathbb{Z}^+\}$. Adicionalmente una vez que reconozca la cadena, queremos que la máquina duplique el número, por lo que debe de agregar un cero al final de la cadena. Dicho esto, tenemos que $Q_1 = \{q_0, q_1, q_a, q_r\}$, $\Gamma_1 = \{0, 1\}$, $\Sigma_1 = \{0, 1, \sqcup\}$ y δ_1 tiene regla de correspondencia como se muestra en la figura 4.2. Supongamos que introducimos la cadena 1010. La configuración inicial de la máquina de Turing será $q_0 1010$.

La máquina de Turing desde la configuración inicial, pasará por las siguientes configuraciones:

$$\sqcup q_0 1010 \sqcup\sqcup, \quad \sqcup 1 q_1 010 \sqcup\sqcup, \quad \sqcup 10 q_1 10 \sqcup\sqcup, \quad \sqcup 101 q_1 0 \sqcup\sqcup, \quad \sqcup 1010 q_a 0 \sqcup.$$

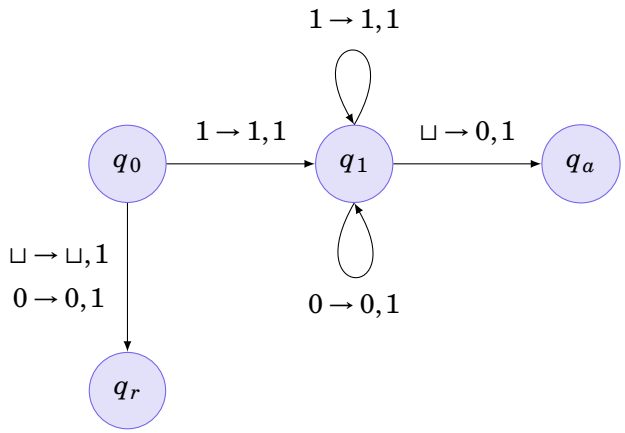


Figura 4.2: Diagrama de flujo de la función de transición δ_1.

Finalmente obtendremos la cadena 10100, la cual es el resultado deseado.

Una forma de ver cómo se comporta la función de transición, es por medio de una digráfica como la de la figura 4.2, perteneciente al ejemplo 4.1. Cuando una máquina de Turing recibe una cadena de Γ^*, hay tres posibles casos: que la máquina acepte a la cadena por medio del estado q_a, que la máquina rechace la cadena por medio del estado q_r o que la máquina itere de manera indefinida. Al diseñar una máquina de Turing, se pretende que no suceda el tercer caso.

Definición 4.2. Sea $M = (Q, \Gamma, \Sigma, \delta)$, la colección de cadenas que acepte una máquina de Turing M es llamado **el lenguaje de** M y se denota por $L(M)$. Decimos que un lenguaje L' es **Turing reconocible** si existe una máquina de Turing M tal que $L' = L(M)$. Si dada cualquier palabra de Γ^*, la máquina siempre llega al estado q_a o q_r, entonces a la máquina se le dice **decisiva**. Sea L un lenguaje del alfabeto Γ, decimos que L es **decidible** si existe una máquina de Turing M tal que para toda palabra en L, la máquina M siempre llegue al estado q_a o q_r.

Las máquinas de Turing deterministas no son las únicas que existen. Existen múltiples tipos de máquinas de Turing, las cuales son llamadas variantes del modelo de Turing. Cabe mencionar que todas las variantes son equivalentes al modelo original. Para los fines de este texto, únicamente definiremos una variante de la máquina determinista, la cual es la máquina de Turing no determinista. Podemos definir al *no determinismo* como ver todas las posibilidades al recibir una instrucción.

Definición 4.3. Definimos una **función de transición no determinista** como $\delta : Q \times \Sigma \to \mathscr{P}(Q \times \Sigma \times \{-1, 1\})$, es decir, dado el estado de la cabecilla y la lectura de la celda en la cinta, podemos ver diferentes posibilidades inmediatas, como cambiar el estado, se puede mover a la izquierda o derecha y sobreescribir en la celda que observe. Definimos a una **maquina de Turing no determinista** N como una tupla $N = (Q, \Gamma, \Sigma, \delta)$ donde Q, Γ y Σ coinciden con la definición 4.1 y δ es una función de transición no determinista.

La única diferencia de las máquinas de Turing deterministas y las no deterministas, es la función de transición. En las primeras se puede ver que la configuración de la máquina se puede representar a partir de una secuencia, para las segundas se tiene

una configuración ramificada. Esto se puede apreciar en las figuras 4.3 y 4.4. Con que sólo una de sus ramas llegue a un estado q_a, la cadena inicial es aceptada. Un resultado importante es que toda máquina de Turing no determinista existe una determinista equivalente [22, pág. 178]. Lo anterior indica que ambos tipos de máquina pueden llegar al mismo resultado, pero posteriormente veremos que no necesariamente al mismo tiempo.

Ejemplo 4.2. Supongamos que tenemos un alfabeto $\Gamma = \{1, \ldots, n\}$, es decir, para cada número natural del 1 al n, tenemos un representante distinto. Sea $Q = \{q_0, q_1, \ldots, q_{n-1}, q_a, q_r\}$. Nuestro objetivo en este ejemplo, es diseñar una máquina de Turing no determinista N todas las palabras $\omega \in \Gamma^*$ de tales que $|\omega| = \text{card}(\Gamma) = n$. Definimos a $\delta : Q \times \Sigma \to \mathscr{P}(Q \times \Sigma \times \{1, -1\})$ dada por

$$\begin{aligned} \delta(q_0, \gamma) &= \{(q_r, \sqcup, 1)\}, \\ \delta(q_i, \sqcup) &= \{(q_{i+1}, 1, 1), (q_{i+1}, 2, 1), \ldots, (q_{i+1}, n, 1)\}, \\ \delta(q_{n-1}, \sqcup) &= \{(q_a, 1, 1), (q_a, 2, 1), \ldots, (q_a, n, 1)\}. \end{aligned}$$

para $\gamma \in \Gamma$ e $i \in \{0, \ldots, n-2\}$. La primer ecuación indica que si lee una cadena distinta a la cadena vacía (ya que $\sqcup \notin \Gamma^*$), la cadena se rechaza automáticamente. En otro caso, el algoritmo enlistará a todas las posibles cadenas de Γ^* con longitud n. Se puede diseñar una máquina de Turing no determinista como la anterior para números binarios y una letra de separación #. En ese caso, $\Gamma = \{0, 1, \#\}$ y la entrada sería el número n codificado en binario, el cual indicaría las posibles combinaciones con repeticiones de longitud n. En este caso, el diseño de la función de transición es más elaborado.

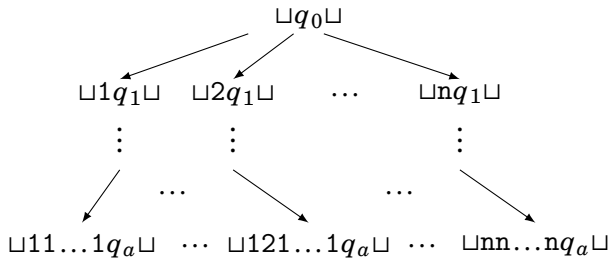


Figura 4.3: Ramificación del algoritmo no determinista del ejemplo 4.2.

Hemos visto que la definición de máquina de Turing sirve como un modelo matemático para definir formalmente un algoritmo. El modelo de máquina de Turing permite recibir las entradas de un algoritmo codificadas en una secuencia de algún alfabeto. Cada vez que se sobrescribe en la cinta, se está iterando el algoritmo. Al llegar a un estado final, la cadena escrita en la cinta es la salida del algoritmo. Hasta ahora hemos definido a las máquinas de Turing a partir de un nivel formal (detallado), al describir los alfabetos, la función de transición, entre otras cosas. A partir de aquí, vamos a utilizar la descripción de alto nivel, la cual solamente se enfoca en describir el algoritmo y sin entrar en los detalles de su implementación abstracta. En el ejemplo 4.3 se da una descripción de alto nivel.

Las estructuras abstractas se pueden interpretar como la palabra de algún lenguaje para realizar cálculos o aplicar algoritmos. Como ejemplo tenemos a la red $R = (\{1,2,3\},\{(1,2),(2,3)\},c)$ con $c(1,2) = 3$ y $c(2,3) = 1/3$, la cual puede ser descrita con la base binaria como

$$\langle R\rangle = \texttt{(1\#10\#11)((1\#10)\#(10\#11))(11\#1/11)},$$

donde $\langle R\rangle$ pertenece a algún lenguaje formado por el alfabeto $\Gamma = \{\texttt{0},\texttt{1},\texttt{(},\texttt{)},\texttt{/},\texttt{\#}\}$. En caso de no incluir a los valores de la función c en la cadena inicial, se puede dar una regla de correspondencia y una máquina de Turing podrá calcular el costo de una arista cuando sea necesario. Recordemos que podemos comparar el orden de dos racionales positivos $a/b, c/d$ con $a,b,c,d \in \mathbb{Z}^+$ y b,d distintos de cero con la definición $a/b < c/d \iff ad < bc$. En general, dado un objeto abstracto A, diremos que $\langle A\rangle$ es su **representación en cinta** o **codificiación** como una cadena formada por algún alfabeto. En caso de contar con los objetos $A_1,\ldots,A_n$, su representación como cadena es denotada por $\langle A_1,\ldots,A_n\rangle$.

4.1.2. Problemas P y NP

Podemos caracterizar el tamaño de una instancia del PAV mediante el número de ciudades que tenga. Para el problema de asignación sucede algo similar, podemos considerar el tamaño de una instancia del problema como la cantidad de tareas que se necesite asignar en dicha instancia. A este tamaño, lo denotaremos como tamaño de la entrada. Para el resto de esta sección, denotaremos a n como el tamaño de la entrada de una máquina de Turing (determinista o no determinista).

Definición 4.4. Sea X un lenguaje de un alfabeto Γ y $Y \subseteq X$. Definimos a un **problema de decisión** como la tupla $\Pi = (X,Y)$ y decimos que el problema es **decidible** si existe una máquina de Turing (un algoritmo) $M = (Q,\Gamma,\Sigma,\delta)$ tal que para toda palabra $\omega_a \in Y$, se llegue al estado q_a y para toda palabra $\omega_r \in X \setminus Y$, se llegue al estado q_r. En este caso decimos que M resuelve a Π.

Un problema de decisión se reduce a saber si una palabra se encuentra en el lenguaje o no. Cuando el lenguaje X no es especificado, se toma a $X = \Gamma^*$, donde Γ es el alfabeto del lenguaje Y. En el caso anterior, nos referimos al lenguaje Y como el problema de decisión. De forma coloquial, los problemas de decisión son aquellos que se responden con afirmaciones o negaciones. A las palabras del lenguaje Y cuando éste es referido como un problema de decisión, se le conocen como **instancias**.

Ejemplo 4.3. Sean $X = \{0,1\}^*$ y L el lenguaje definido en el ejemplo 4.2. Sea $m \in \mathbb{N}$ y $\langle m\rangle$ su representación en base 2. Definimos la siguiente máquina de Turing M:

M = Recibe $\langle m\rangle$, el número m codificado en base dos:

1. Se observa la primera entrada de $\langle m\rangle$.

2. Si la primera letra de la cadena $\langle m\rangle$ es 1, *aceptamos*. En otro caso, *rechazamos*.

La máquina de Turing M muestra que el problema $\Pi = (X,L)$ es decidible. Si introducimos la cadena 1010, entonces en el segundo paso se acepta a la entrada.

Definición 4.5. Sea M una máquina de Turing decisiva. Definimos **el número de pasos de ejecución de** M como una función $f : \mathbb{N} \to \mathbb{N}$, donde $f(n)$ es el máximo número de pasos que M utiliza en total (si es determinista) o como la cardinalidad de la rama más larga (si es no determinista) para una entrada de tamaño n. Veamos la figura 4.4. Notemos que podemos definir una función monótona no decreciente $T : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ tal que $T \circ f = t$ donde $t : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ es el tiempo de ejecución de un algoritmo.

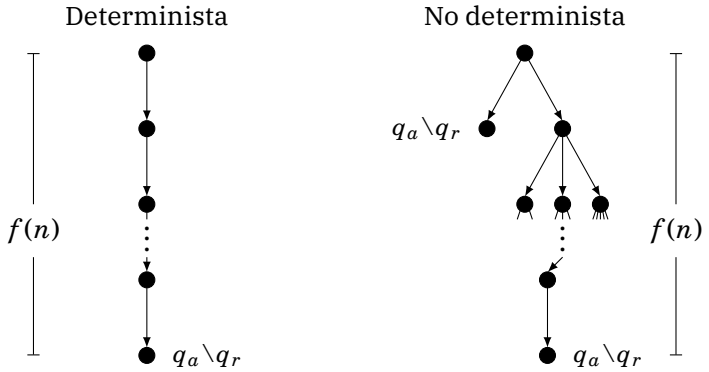


Figura 4.4: Pasos de ejecución para máquinas deterministas y no deterministas.

La máquina determinista del ejemplo 4.1 necesitó $n+1$ pasos para su ejecución, con n el tamaño del número binario que recibía al inicio. Además, con la del ejemplo 4.2 pasa algo similar, es no determinista y todas sus ramas tienen n pasos (sin contar la configuración inicial), entonces tiene $n+1$ pasos de ejecución, con n el tamaño de las posibles configuraciones a encontrar.

Definición 4.6. Sea $t : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ una función. Definimos **la clase de complejidad de tiempo determinista**, $\text{TIME}(t(n))$, como el conjunto de lenguajes que son decidibles en tiempo $\mathcal{O}(t(n))$ por alguna máquina de Turing determinista. Definimos la clase de problemas **polinomiales (P)** como la clase de lenguajes que son decidibles en tiempo polinomial por una máquina de Turing determinista. Matemáticamente, esto sería

$$\text{P} = \bigcup_{k \in \mathbb{N}} \text{TIME}\left(n^k\right).$$

Continuando con el ejemplo 4.3, veamos si el problema de decisión Π está en P. Dada $\omega \in \{0,1\}^*$, sólo es necesario ver la primera letra de la cadena para verificar que efectivamente es una palabra en L. Esto implica que el lenguaje es decidible en tiempo $\mathcal{O}(k)$ para alguna constante k no negativa, pues sin importar el tamaño de la entrada, el tiempo de verificación siempre será idéntico.

Definición 4.7. Sea $t : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ una función. Definimos **la clase de complejidad de tiempo no determinista**, $\text{NTIME}(t(n))$, como el conjunto de lenguajes que son decidibles en tiempo $\mathcal{O}(t(n))$ por alguna máquina de Turing no determinista. Definimos la clase de problemas **no deterministas polinomiales (NP)** como la clase de lenguajes que son decidibles en tiempo polinomial por una máquina de Turing no determinista. Matemáticamente, esto sería

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}\left(n^k\right).$$

Proposición 4.8. *La clase de problemas* P *está contenida en* NP (P ⊆ NP)*.*

Demostración. Sean $\Pi \in \mathrm{P}$ un problema de decisión y M una máquina determinista que lo resuelva en tiempo polinomial. Como toda máquina de Turing determinista es no determinista, ya que $Q \times \Sigma \times \{-1,1\} \in \mathscr{P}(Q \times \Sigma \times \{-1,1\})$, entonces M es una máquina no determinista tal que resuelve a Π y por lo tanto, $\Pi \in \mathrm{NP}$. □

Una propiedad importante que cumplen las máquinas de Turing deterministas, es que para toda máquina de Turing no determinista decisiva que tarde un tiempo $t(n)$ en decidir, existe una máquina de Turing determinista decisiva equivalente que tarda un tiempo $2^{\mathscr{O}(t(n))}$ en decidir [22, pág. 284]. Otra propiedad importante es que los lenguajes NP son aquellos que se pueden verificar en tiempo polinomial.

Definición 4.9. Un **verificador** de un lenguaje L sobre un alfabeto Γ es una máquina de Turing decisiva V tal que se puede definir a L como

$$L = \{\omega \in \Gamma^* : V \text{ acepta a la cadena } \langle \omega, c \rangle, \text{ para alguna cadena } c \in \Gamma^*\}.$$

A la cadena c se le suele llamar **certificado** o **testigo** y se utiliza para comprobar que $\omega \in L$. Pediremos adicionalmente que $|c| \leq p(|\omega|)$, donde $p : \mathbb{N} \to \mathbb{N}$ es un polinomio. Esta definición se ejemplifica más adelante.

4.1.3. Problemas NP-Completos y NP-Duros

Ya hemos definido las clases de problemas P y NP, además de haber visto algunos ejemplos y propiedades. En palabras simples, los problemas P son aquellos que son rápidos de decidir y los problemas NP son rápidos en verificar. Una pregunta importante que hasta la fecha sigue sin tener respuesta, es saber si P está estrictamente contenido en NP o son iguales. Un gran avance fue desarrollado por Stephen Cook y Leonid Levin en los años 70's, al definir una clase de problemas llamados no deterministas polinomiales completos (NP – C). Antes de definir a los problemas NP – C, veamos el primer problema conocido del conjunto NP – C.

El problema de satisfacibilidad booleana (SAT) consiste en saber si para una fórmula lógica de la forma normal conjuntiva (de la definición 4.10), existe una combinación de ceros y unos tal que resulte ser verdadera. Dicha fórmula lógica, sólo contiene a los operadores *y, o* y *no*, representados por los símbolos ∧, ∨ y ¬ respectivamente.

Definición 4.10. Definimos al operador unario $\neg : \{0,1\} \to \{0,1\}$ y a los operadores binarios $\wedge, \vee : \{0,1\}^2 \to \{0,1\}$ con las reglas de correspondencia $\neg x = 1 - x$, $x \wedge y = \min\{x, y\}$ y $x \vee y = \max\{x, y\}$. Dado $n \geq 2$ un natural, definimos a los operadores $\bigwedge, \bigvee : \{0,1\}^n \to \{0,1\}$ con las reglas de correspondencia

$$\bigwedge(x_1, \ldots, x_n) = \min\{x_1, \ldots, x_n\}, \quad \bigvee(x_1, \ldots, x_n) = \max\{x_1, \ldots, x_n\}.$$

Decimos que la expresión booleana $\phi : \{0,1\}^n \to \{0,1\}$ está en **forma normal conjuntiva (FNC)** si su expresión está dada en la forma

$$\phi(x) = \bigwedge_{i \in I} \left(\bigvee_{j \in J_i} \tau_j \right),$$

donde $I \subseteq \mathbb{Z}^+$ es un conjunto finito, $J_i \subseteq \{1,\dots,n\}$ para toda $i \in I$ y $\tau_j = x_j$ o $\tau_j = \neg x_j$. La expresión ϕ es **satisfacible** si existe $x \in \{0,1\}^n$ tal que $\phi(x) = 1$.

Ejemplo 4.4. Sea $\phi_1 : \{0,1\}^4 \to \{0,1\}$ la expresión booleana con la regla de correspondencia $\phi_1(x_1,x_2,x_3,x_4) = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_4)$. Notemos que $(0,0,0,1)$ satisface a ϕ_1, ya que

$$\begin{aligned}\phi_1(0,0,0,1) &= \text{mín}\{\text{máx}\{1-0,0\},\text{máx}\{1-0,1\}\}\\ &= \text{mín}\{\text{máx}\{1,0\},\text{máx}\{1,1\}\}\\ &= \text{mín}\{1,1\} = 1.\end{aligned}$$

Algo conveniente de mencionar de la definición 4.10, es que a las τ_j's se les conoce como los **términos** de las variables x_j's y a cada componente $\bigvee_{j\in J_i} \tau_j$ como una **cláusula**.

Definición 4.11. Definimos al **problema de satisfacibilidad booleana (SAT)** como el problema de decisión $\Pi = (X,Y)$ tal que

$$\begin{aligned}X &= \{\langle\phi\rangle : \phi \text{ está en FNC}\},\\ Y &= \{\langle\phi\rangle : \phi \text{ está en FNC y es satisfacible}\}.\end{aligned}$$

Es decir, si para una fórmula en FNC ϕ encontramos una combinación x de ceros y unos tal que $\phi(x) = 1$, aceptamos a ϕ. Regresando al ejemplo 4.4, notemos que la cadena $c = 0001$ es un certificado para ϕ_1, ya que podemos diseñar algún verificador V que compruebe que $\langle\phi_1\rangle$ es satisfacible, por medio de $\langle\phi_1, c\rangle$. El SAT resulta ser de los problemas más difíciles de NP, la pregunta es ¿cómo podemos definir a los problemas difíciles de NP?

Definición 4.12. Dados dos lenguajes L_1 y L_2 del alfabeto Γ, una función $f : L_1 \to L_2$ es una **función computable en tiempo polinomial** si existe una máquina de Turing determinista decisiva M que se detenga cuando la cadena $f(\omega) \in L_2$ esté en su cinta, dada la cadena inicial $\omega \in L_1$. Un lenguaje A es un **polinomialmente reducible** a un lenguaje B (denotado por $A \leq_P B$) si existe una función computable en tiempo polinomial $f : \Gamma^* \to \Gamma^*$, donde para toda $\omega \in \Gamma^*$, $\omega \in A$ si y sólo si $f(\omega) \in B$. Lo anterior se puede generalizar para problemas de decisión, tomando a los problemas $\Pi_1 = (X_1,Y_1)$ y $\Pi_2 = (X_2,Y_2)$ sobre un alfabeto Γ, decimos que $\Pi_1 \leq_P \Pi_2$ si y sólo si $f : X_1 \to X_2$ es una función computable en tiempo polinomial tal que para todo $\omega \in X_1$, $\omega \in Y_1$ si y sólo si $f(\omega) \in Y_2$.

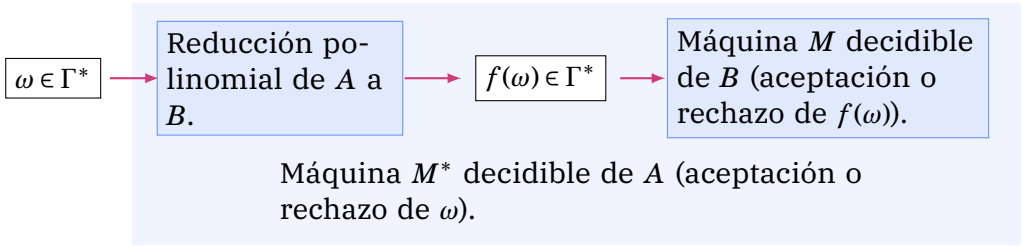


Figura 4.5: Concepto de reducción polinomial.

Consideremos a $\Pi_1, \Pi_2 \in$ NP. Si tenemos un método eficiente para resolver a Π_1 y una reducción polinomial de Π_2 a Π_1, entonces podemos resolver eficientemente a Π_2. De lo

anterior, podríamos decir que Π_2 es a lo más tan difícil de resolver que Π_1. Si cualquier $\Pi \in \text{NP}$ es reducible en tiempo polinomial a $\Pi^* \in \text{NP}$, entonces Π^* sería el más difícil de resolver dentro de NP, o uno de los más difíciles de resolver, en caso de que no fuera el único. La siguiente definición nos permite conocer a los problemas más difíciles de NP.

Definición 4.13. Un lenguaje B es **NP-Completo (o NP−C)** si se satisface que

1. $B \in \text{NP}$;
2. para todo $A \in \text{NP}$, $A \leq_{\text{P}} B$.

A la segunda condición de la definición 4.13 se le conoce como **NP−completez**. El primer problema conocido del conjunto NP – C fue el SAT y esto fue demostrado por Stephen Cook y Leonid Levin en el año 1971 [22, pág. 299]. A este resultado se le conoce como el Teorema de Cook–Levin. Otro resultado importante es que dados los lenguajes A y B sobre un alfabeto Γ tales que $A \in \text{NP} - \text{C}$, $B \in \text{NP}$ y $A \leq_P B$, entonces $B \in \text{NP} - \text{C}$ [22, pág. 304]. Adicionalmente, Richard Karp exhibió en 1972, por medio del Teorema de Cook-Levin y del resultado anterior, que había 21 problemas dentro del conjunto NP – C [34].

Para este punto, tenemos las herramientas necesarias para demostrar que el problema de decisión asociado al PAV, es NP – C. Primero demostraremos que el problema es NP, para ello nos basaremos en una demostración del libro *Introduction to the Theory of Computation* de Sipser [22]. Posteriormente demostraremos por medio del SAT que el problema de decisión asociado al PAV está en NP – C, ya que el SAT es NP – C y el SAT puede ser reducido a dicho problema de decisión, tendríamos que nuestro problema es NP – C [35]. La reducción en tiempo polinomial se realiza al convertir fórmulas FNC a redes cuyos ciclos hamiltonianos estén asociados a un vector $x \in \{0,1\}^n$ que satisfaga la fórmula en FNC.

Teorema 4.14. *Consideremos el lenguaje $L \subseteq \Gamma^*$, para algún alfabeto Γ, cuyas palabras son de la forma $\langle R, k\rangle$, donde $R = (X, A, c)$ una red dirigida completa y sin bucles, $c(a)$ es polinomialmente computable para cualquier $a \in A$ y k es un racional no negativo tal que $|\langle k\rangle| \leq p(|\langle R\rangle|)$ para algún polinomio $p : \mathbb{N} \to \mathbb{N}$, donde se satisface que R contiene al menos un circuito hamiltoniano t con $c(t) \leq k$. El lenguaje L es* NP – C.

Demostración. P.d. El problema es NP. Para esto necesitamos, construir una máquina de Turing no determinista que resuelva el problema en tiempo polinomial. Consideremos a $n := \text{card}(X)$ y que $X = \{1, \ldots, n\}$. Definimos a la máquina de Turing no determinista N con la siguiente descripción de alto nivel:

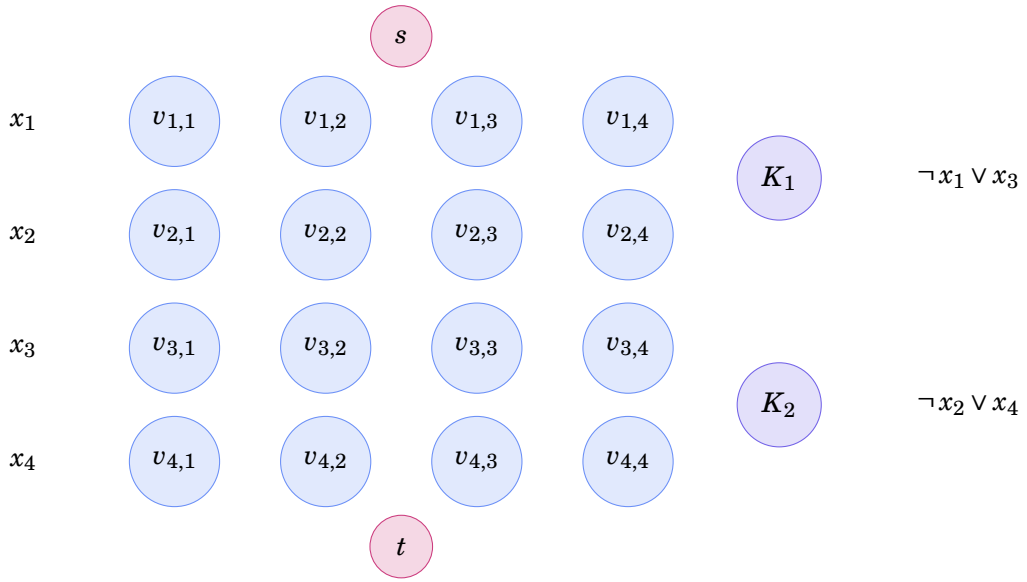
N = Recibe la cadena $\langle R, k\rangle$, donde R es una red dirigida, completa y sin bucles y k es un real no negativo computable en tiempo polinomial.

1. Escribir una lista de n números naturales, $r_1, \ldots, r_n$, tales que $1 \leq r_i \leq n$, para $i \in \{1, \ldots, n\}$. Cada número en la lista es seleccionado de manera no determinista.
2. Verificar si hay números repetidos en la lista. En el caso de encontrar repeticiones, *rechazamos*.
3. Verificar que $\sum_{i=1}^{n-1} c(r_i, r_{i+1}) + c(r_n, r_1) \leq k$. Si no se satisface la desigualdad anterior, *rechazamos*. En otro caso, *aceptamos*.

Vemos que en paso 1, se exponen todos los posibles ordenamientos (con repeticiones) de la lista $\{1,\dots,n\}$. Aquí se puede aplicar la máquina de Turing no determinista vista en el ejemplo 4.2. Dicho algoritmo necesita de n pasos para su ejecución. Para el paso 2, revisar si hay al menos un número repetido. Finalmente, en el paso 3 se verifica la desigualdad. Todo lo anterior se puede realizar en tiempo polinomial no determinista. Por lo tanto, el problema está en NP.

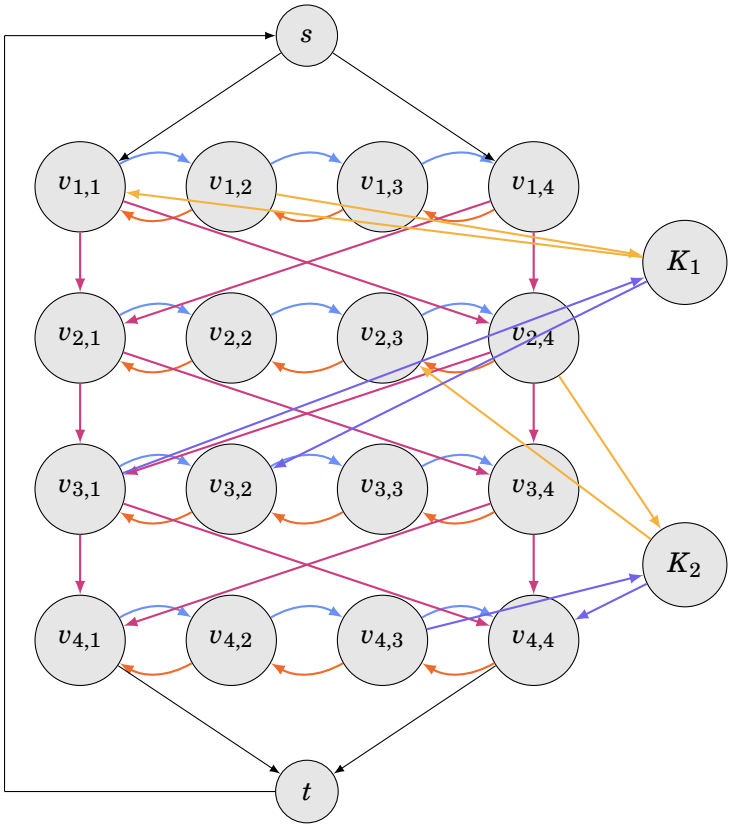
P.d. El SAT es reducible a nuestro problema en tiempo polinomial. Para toda instancia del SAT, podemos construir una red $R = (X, A, c)$. Vamos a demostrar que una fórmula en FNC es satisfacible si y sólo si existe un tour t en la red R tal que $c(t) \leq k$, para alguna $k \in \mathbb{Q}$ que cumpla la hipótesis del teorema. Consideremos a ϕ una fórmula con n variables. Sea m el número de cláusulas. Nuestro objetivo es construir una red dirigida $R = (X, A, c)$ donde al resolver el problema del enunciado a demostrar, obtendremos un vector $(x_1,\dots,x_n) = x$ tal que $\phi(x) = 1$. A la par de dar la construcción general, veremos la construcción para el ejemplo 4.4 de manera visual.

Consideremos n conjuntos de nodos diferentes $C_1,\dots,C_n$ correspondientes a las n variables de nuestra fórmula ϕ. Cada conjunto está formado por $2m$ nodos, es decir, $C_i = \{v_{i,1}, v_{i,2},\dots,v_{i,2m}\}$ para $i \in \{1,\dots,n\}$. Además de estos nodos, agregaremos un nodo por cada cláusula de ϕ, es decir, los nodos $K_1,\dots,K_m$. Finalmente agregaremos los nodos s y t. Observemos que $X = C_1 \cup \cdots \cup C_n \cup \{K_1,\dots,K_m, s, t\}$. Para $\phi_1(x) = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_4)$, tenemos dos cláusulas y cuatro variables, por lo que agregaremos los siguientes nodos:



El siguiente paso es definir al conjunto A de nuestra red. Agregamos a los arcos $(v_{i,j}, v_{i,j+1})$ para $i \in \{1,\dots,n\}$ y $j \in \{1,\dots,2m-1\}$ (señalados con color azul), los cuales indican que debemos igualar a 1 la variable x_i si son recorridos. También agregaremos a los arcos $(v_{i,j}, v_{i,j-1})$ para $i \in \{1,\dots,n\}$ y $j \in \{2,\dots,2m\}$ (señalados con color naranja), los cuales indican que debemos igualar a 0 la variable x_i si son recorridos. Vamos a conectar los conjuntos C_i por medio de los arcos $(v_{i,1}, v_{i+1,1})$,$(v_{i,1}, v_{i+1,2m})$, $(v_{i,2m}, v_{i+1,1})$ y $(v_{i,2m}, v_{i+1,2m})$ para $i \in \{1,\dots,n-1\}$ (estos arcos están señalados con color fucsia). Para conectar a los arcos s y t con las cadenas C_i's, agregaremos a los arcos $(s, v_{1,1})$,$(s, v_{1,2m})$,

$(v_{n,1}, t)$, $(v_{n,2m}, t)$ y (t, s). Finalmente conectaremos a los nodos K_i's con el resto de la red. Dadas $i \in \{1, \ldots, n\}$ y $j \in \{1, \ldots, m\}$, si x_i es el término de la i-ésima variable en la j-ésima cláusula de ϕ, entonces agregamos los arcos $(v_{i,2j-1}, K_j)$ y $(K_j, v_{i,2j})$ (señalados con color lila); o si $\neg x_i$ es el término de la i-ésima variable en la j-ésima cláusula de ϕ, entonces agregamos los arcos $(K_j, v_{i,2j-1})$ y $(v_{i,2j}, K_j)$ (señalados con color amarillo). Veamos el ejemplo:



Como en nuestro problema de decisión la red debe ser completa y sin bucles, agregamos los arcos restantes a R. Finalmente definimos a la función $c : A \to \mathbb{R}$ con la regla de correspondencia $c(a) = k/(2mn + m + 2)$ para los arcos a definidos en el párrafo anterior y $c(a) = 2k$ para los demás arcos, la cual es computable en tiempo polinomial. A continuación, unas observaciones para continuar con la demostración.

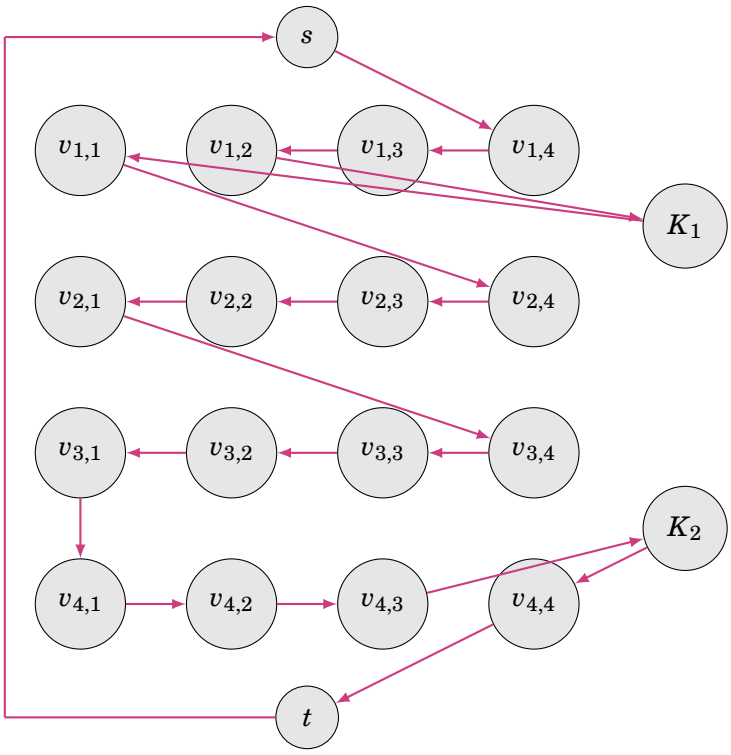
- Esta red es construida en tiempo polinomial.
- Como $\text{card}(X) = 2mn + m + 2$ y todo ciclo hamiltoniano en una digráfica es de la misma cardinalidad, entonces los tours t que sólo contengan arcos con costo menor a k, son tours de costo $c(t) = k$.
- Cualquier ciclo hamiltoniano de costo k pasa de derecha a izquierda o de izquierda a derecha por los conjuntos C_i's. Si tenemos un tour de costo k, se debe de ir de un nodo $v_{i,j}$ a un nodo $v_{i+1,j}$ directamente o cruzando por algún nodo K. De forma similar, podemos ir de un nodo $v_{i-1,j}$ a un nodo $v_{i,j}$.
- Suponiendo que existe un vector que satisfaga la fórmula ϕ, recorremos el conjunto C_i de izquierda a derecha si $x_i = 1$, o en sentido contrario en otro caso. Se

debe recorrer la cláusula cuando sea posible. Debemos conectar a los C_i's de tal forma que se mantenga la continuidad del ciclo. Al final, conectamos a un nodo de C_n con t, a t con s y a s con un nodo de C_1. Al concluir, obtendremos un ciclo hamiltoniano en R de costo k. Lo anterior satisface la condición $\omega \in A \Rightarrow f(\omega) \in B$ de la definición 4.12.

- Suponiendo que existe un tour t de costo k en la gráfica, si atravesamos el conjunto C_i de izquierda a derecha, entonces asignamos a $x_i = 1$, en otro caso, $x_i = 0$. Dado que t visita una vez cada nodo K_j, al menos un conjunto C_i atravesó al nodo K_j en la dirección adecuada. Al concluir, obtendremos una asignación que satisface la fórmula ϕ. Lo anterior satisface la condición $f(\omega) \in B \Rightarrow \omega \in A$ de la definición 4.12.

- Nótese que también queda demostrado que no existe un ciclo hamiltoniano de presupuesto a lo más k si y sólo si la fórmula ϕ es no sasfacible.

Con los argumentos anteriores, queda demostrado que el problema de decisión asociado al PAV es NP – C. Con el certificado del ejemplo 4.4, podemos ver que un ciclo hamiltoniano asociado a él es el siguiente



□

Hasta el momento no hemos hablado concretamente de los problemas de optimización. Así como los problemas NP son fáciles de verificar, los problemas NPO forman un conjunto de problemas de optimización cuyas soluciones factibles de una instancia dada son fáciles de comparar. Estos problemas serán definidos a continuación desde una perspectiva de optimización combinatoria. También definiremos a su óptimo y a

su valor óptimo. De aquí, resta extender la noción de dificultad a problemas de optimización y finalmente demostrar que el PAV es un problema *difícil*. Para ello, haremos uso del libro *Complexity and Approximation* de Ausiello, Crescenzi, Gambosi y col. [36].

Definición 4.15. Decimos que una tupla $\Omega = (X, S, z, m)$ es un **problema de optimización** donde

1. X es el **conjunto de entradas** o **instancias** codificadas en algún lenguaje L con alfabeto Γ;
2. $S : X \to \Gamma^*$ es la **función de soluciones factibles**, la cual asocia a una instancia x con su conjunto de soluciones factibles $S(x)$;
3. $z : \{(x, y) : x \in X, y \in S(x)\} \to \mathbb{Q}$ es la **función objetivo**;
4. $m \in \{\text{Min}, \text{Max}\}$ es la **dirección de optimización**.

Ejemplifiquemos la definición anterior por medio del PAV. Recordemos que en el capítulo 2 vimos cómo modelar a las instancias del problema de distintas formas. Una forma conveniente es por medio de una matriz C como la de la ecuación 2.5, entonces las instancias del PAV serían de la forma $x = \langle C \rangle$. Suponiendo que n es el número de ciudades de nuestra instancia, cada solución factible se pueden representar con una permutación de orden n tal que $\sigma(1) = 1$, de manera que nuestro conjunto de soluciones factibles sería de la forma $S(x) = \langle \sigma_1, \sigma_2, \ldots, \sigma_{(n-1)!} \rangle$. Dadas $x \in X$, una matriz de costos C, y $y \in S(x)$ una permutación σ donde ambas están codificadas, la función objetivo queda definida como $z(x, y) = c_{\sigma(1),\sigma(2)} + c_{\sigma(2),\sigma(3)} + \cdots + c_{\sigma(n),\sigma(1)}$. Finalmente, en el PAV se desea encontrar el tour de menor costo, por lo que la dirección de optimización es $m = \text{Min}$.

Definición 4.16. Decimos que un problema de optimización $\Omega = (X, S, z, m)$ es un **problema no determinista polinomial de optimización (NPO)** si

- X es Turing reconocible en tiempo polinomial determinista;
- para toda $y \in S(x)$ se satisface que $|y| \leq p(|x|)$ con $p : \mathbb{N} \to \mathbb{N}$ un polinomio;
- $S(x)$ es un lenguaje Turing reconocible en tiempo polinomial determinista;
- La función objetivo z es computable en tiempo polinomial determinista.

Así como las palabras en los problemas de decisión NP son rápidas de verificar, las soluciones factibles de instancias en problemas de optimización NPO son rápidas de comparar. Sea $x \in X$ una instancia de Ω, a partir de la definición anterior podemos ver que dadas $y_1, y_2 \in S(x)$, es sencillo realizar una comparación, ya que $z(x, y_1)$ y $z(x, y_2)$ se puede calcular fácilmente y dependiendo de la dirección de optimización de Ω, sabremos qué alternativa es mejor. En el caso del PAV, podemos caracterizar a una instancia de n ciudades con una matriz de $n \times n$, lo cual es polinomial. Las soluciones factibles se pueden representadar por una permutación de orden n o una matriz de ceros y unos, lo cual también es polinomial. Finalmente, evaluar la función objetivo también es un procedimiento polinomial, ya que se realizan $n(n-1)$ multiplicaciones y posteriormente $n(n-1)-1$ sumas.

Definición 4.17. Sea $\Omega = (X, S, z, m)$ un problema de optimización. Dada una instancia $x \in X$, decimos que y_x^* es su **óptimo** si $z(x, y_x^*) = m\{z(x, y) : y \in S(x)\}$ y su **valor óptimo** es $\text{opt}(x) = z(x, y_x^*)$.

Dado un problema de optimización, derivan tres problemas diferentes mediante los cuales se puede llegar a una solución al resolverlos. La primera es obtener la solución factible que al evaluar en la función objetivo, nos dé el mejor valor. La segunda forma es obtener el mejor valor óptimo. La tercera forma es saber si dado un presupuesto k, existe una solución factible tal que su valor en la función objetivo sea al menos tan bueno como k.

Definición 4.18. Dado un problema de optimización $\Omega = (X, S, z, m)$, definimos a los siguientes problemas:

- **problema de construcción** Ω_C, dada una instancia $x \in X$, se debe llegar al óptimo y_x^* y a su valor óptimo $\text{opt}(x)$;
- **problema de evaluación** Ω_E, dada $x \in X$, se debe obtener al valor óptimo $\text{opt}(x)$;
- **problema de decisión** Ω_D, dada $x \in X$ y k un real no negativo computable, verificar si existe $y \in S(x)$ tal que $z(x, y) \leq k$ si $m = \text{Min}$ y $z(x, y) \geq k$ en otro caso.

Un algoritmo que resuelve el problema de construcción del PAV, es el de ramificación y acotamiento, expuesto en el capítulo 2. Sólo nos enfocaremos en los de construcción y de decisión para desarrollar lo restante. Algunos lenguajes L son tan difíciles que aunque satisfagan la NP−completez, no necesariamente están en NP [22, pág. 325]. Además, la definición de reducibilidad polinomial no se adapta a los problemas de construcción, por lo que necesitamos una definición de reducibilidad más general.

Definición 4.19. Definimos al **oráculo de un lenguaje** B como un dispositivo externo capaz de responder si una cadena $\omega \in B$. Definimos al **oráculo de un problema de construcción de** $\Omega = (X, S, z, m)$ como un dispositivo externo capaz de determinar el óptimo y_x^* y su valor óptimo $\text{opt}(x)$ para cualquier instancia $x \in X$. Se dice que una máquina de Turing M es una **máquina de Turing con oráculo**, si M tiene la capacidad de preguntarle a un oráculo. Denotaremos a ésta con M^{Π}.

En síntesis, un oráculo es un objeto abstracto que nos permite conocer la respuesta a un problema. Usualmente se le pide al oráculo que dé la respuesta de manera inmediata, por lo que si M^{Π} es una máquina de Turing con oráculo para el problema Π, por lo que M^{Π} resuelve el problema en un paso.

Definición 4.20. Sean Π_1 y Π_2 problemas (de decisión o construcción). Decimos que Π_1 es **Turing reducible polinomial** (o simplemente **Turing reducible**) a Π_2, denotado por $\Pi_1 \leq_T \Pi_2$, si existe máquina de Turing determinista M con acceso a un oráculo de Π_2 que resuelva a Π_1 en tiempo polinomial. Si $\Pi_1 \leq_T \Pi_2$ y $\Pi_2 \leq_T \Pi_1$, entonces decimos que Π_1 y Π_2 son **Turing equivalentes** y se denota por $\Pi_1 \equiv_T \Pi_2$. Decimos que Π_1 es **NP−Duro (**$\text{NP} - \text{D}$**)** si para todo $\Pi \in \text{NP}$, se satisface que $\Pi \leq_T \Pi_1$.

Así como la reducibilidad polinomial, la Turing reducibilidad nos permite saber qué tan difícil es un problema a comparación de otro. Si para dos problemas Π_1 y Π_2 sabemos que $\Pi_1 \leq_T \Pi_2$, por consiguiente resolver Π_1 no es tan complicado como resolver Π_2. Dicho eso, si un problema es NP−Duro, entonces es al menos tan difícil de resolver como cualquier problema en NP. Además, si dos problemas son Turing equivalentes, uno es tan difícil de resolver como el otro. Para demostrar que un problema es NP−Duro, es suficiente con demostrar que un problema NP−Completo es Turing reducible a él.

Teorema 4.21. *El problema de construcción del agente viajero es* NP – D.

Demostración. Sea $\Omega = (X, S, z, \text{Min})$ el problema del agente viajero, Ω_C su problema de construcción y Ω_D su problema de decisión (definido en la proposición 4.14). Tomamos $x \in X$ una instancia de Ω (una red) y $\langle x, k\rangle$ una instancia de Ω_D asociada a x, con k un real no negativo tal que $|\langle k\rangle| \leq p(|\langle x\rangle|)$ para algún polinomio $p : \mathbb{N} \to \mathbb{N}$. P.d. $\Omega_D \leq_T \Omega_C$. Al tener un oráculo para Ω_C, sólo necesitaríamos comparar a la constante k de la instancia $\langle x, k\rangle$ con el valor óptimo $\text{opt}(x)$ de la instancia x. Si $\text{opt}(x) \leq k$, entonces aceptamos a la instancia $\langle x, k\rangle$, en otro caso, rechazamos. Por lo tanto $\Omega_D \leq_T \Omega_C$. Como Ω_D es NP−C, queda demostrado que el problema del agente viajero es NP−Duro.

□

Para finalizar con esta sección, es importante mencionar que dado un problema de optimización, los problemas asociados de decisión, evaluación y construcción son Turing equivalentes, por lo que los tres tienen el mismo nivel de dificultad [36, págs. 31−32]. Hasta la fecha, no se conoce algún algoritmo que resuelva el problema de construcción para el PAV en tiempo polinomial, ni se ha demostrado que dicho algoritmo no exista. En caso de que exista un algoritmo que resuelva en tiempo polinomial el PAV, los conjuntos P y NP serían iguales, en caso de no existir, sabríamos que P está estrictamente contenido en NP. Desde 1971, Steven Cook planteó la conjetura P v.s. NP y aún en la actualidad, no se ha logrado responder. El problema es tan relevante que el Instituto Clay de Matemáticas ofrece una compensación de un millón de dólares a la persona o las personas que resuelvan la conjetura [37].

4.2. Una heurística para el PAV

Ya hemos visto que el problema del agente viajero es un problema difícil y esto es técnicamente imposible al resolver grandes instancias. En general, al no contar con algoritmos eficientes para los problemas NP−Duros de optimización, se opta por métodos que no necesariamente llegarán al optimo del problema, pero se ejecutan en tiempo polinomial. En el resto del capítulo, nos enfocaremos en dar una síntesis de ellos, llamados heurísticos, y ver una implementación de uno para el PAV.

4.2.1. Las heurísticas como alternativas de optimización

A estas alturas del texto, es preciso hacer una reflexión sobre la metodología empleada para el desarrollo de las matemáticas. Al momento de enunciar un resultado matemático, es necesaria su demostración para ser aceptado como verdadero, como explica Polya [38, págs. 215−221] en *How to solve it*. Esto nos permite que la teoría matemática tenga bases sólidas y resultados fuertes. Como ejemplo de lo anterior, tenemos al famoso teorema de Pitágoras. El enunciado nos dice que dado un triángulo rectángulo, la suma de los cuadrados de sus catetos es igual al cuadrado de su hipotenusa. No se sabe con certeza si alguien de la escuela pitagórica lo demostró, pero el teorema es publicado como la proposición 47 del libro *los Elementos* de Euclides, donde se demuestra de dos formas distintas.

Mansfield y Wildberger [39] comentan que en el siglo pasado, el arqueólogo Edgar Banks descubrió una tabla a la que se bautizó con el nombre de *Plimpton 322*, la cual fue

tallada por los babilonios. En dicha tabla se encuentran escritas ternas pitagóricas, que son números enteros que satisfacen la ecuación $a^2 + b^2 = c^2$. De aquí podemos ver que aunque los pitagóricos y Euclides contaban con un enunciado formal, siglos antes los babilónicos *descubrieron* soluciones enteras a la ecuación pitagórica. Esta es la esencia de la metodología heurística, la cual no necesariamente lleva a la solución definitiva de un problema, pero lleva al descubrimiento de soluciones provisionales, prácticas y útiles [38, págs. 112-114]. Al no saber si existen algoritmos eficientes para resolver ciertos problemas de optimización bajo el modelo computacional de Turing, los métodos heurísticos son una alternativa conveniente para obtener buenas soluciones en poco tiempo.

En optimización, la metodología heurística es empleada para *resolver* diversos problemas NP – D. Según Wang y Chen [40], una **heurística** o **algoritmo heurístico** es un método computacional que intenta mejorar alguna solución a partir de la medida de optimalidad. En nuestro contexto, la heurística trata de determinar mejores soluciones factibles $y \in S(x)$ de manera iterativa en tiempo eficiente para una instancia $x \in X$ de algún problema de optimización $\Omega = (X, S, z, m)$, el cual usualmente es NPO por su facilidad de comparación. Dependiendo de los autores, a los algoritmos heurísticos también se les suele llamar **metaheurísticas** [41], donde el prefijo *meta* hace referencia a algo más allá de las heurísticas.

Aparte de lo ya comentado, Wang y Chen [40] mencionan que otra ventaja usual de los algoritmos heurísticos es que no necesitan demasiada memoria para su ejecución. Notemos que el algoritmo de ramificación y acotamiento del capítulo 2, el cual no es heurístico, no cuenta con esta virtud, pues la memoria utilizada tiende a crecer al menos de manera exponencial en función del número de ciudades. Ahora, hay dos principales problemas con esta metodología. Uno ya mencionado es que estos métodos no garantizan encontrar la solución óptima sin buscar por todo el conjunto de soluciones factibles. El otro es que en algunos casos no tendremos garantía de que nos *acercaremos* al óptimo en un tiempo razonable.

Melian, Moreno-Pérez y Moreno-Vega [41] proponen características que nos gustaría tener en un algoritmo heurístico. Dependiendo el problema o el contexto, ciertas propiedades serán más *deseables* que otras. Algunas características para el algoritmo son las siguientes: 1) general: el método puede ser utilizado para amplia variedad de problemas e instancias; 2) efectivo: debe proporcionar soluciones próximas al óptimo; 3) eficaz: la probabilidad de llegar al óptimo en casos prácticos debe ser próxima a 1; 4) eficiente: debe aprovechar los recursos computacionales y ejecutarse en poco tiempo; 5) simple: debe ser sencillo de comprender; 6) robusto: no debe ser sensible a pequeñas alteraciones en la instancia.

Algunos tipos de heurísticas que mencionan, son los siguientes:

1. Las heurísticas de **relajación** resuelven versiones fáciles de los problemas a optimizar al retirar restricciones del modelo.

2. Las heurísticas **constructivas** van añadiendo componentes a la solución a partir de un proceso de análisis y selección.

3. Las heurísticas de **búsqueda** definen un concepto de vecindad con el fin de encontrar mejores soluciones a partir de la medida de optimalidad.

4. Las heurísticas **evolutivas** se basan en la evolución de un conjunto de soluciones, el cual tiende a mejorar en cada iteración.

Una heurística de relajación para el PAV sería el algoritmo húngaro del capítulo 2, la cual no proporciona soluciones factibles, pero sí cotas inferiores para el valor óptimo. Los algoritmos de glotones son algoritmos constructivos. Un método de este tipo es Kruskal [17, págs. 9–10]. Un ejemplo de algoritmo de búsqueda es el algoritmo simplex [13, pág. 82], pero éste no es un heurístico. Los algoritmos genéticos constituyen un ejemplo de algoritmos evolutivos, los cuales son presentados a continuación.

4.2.2. Algoritmos genéticos

A los algoritmos cuya cadena final dependa de números aleatorios son conocidos como **algoritmos aleatorios**, según Tardos y Kleinberg [42]. Es necesario hacer énfasis en que los algoritmos genéticos involucran aleatoriedad para su ejecución, al igual que otras heurísticas. Alves da Silva y Falcão [43] comentan que los algoritmos genéticos son una alternativa eficiente para resolver problemas de gran escala, como lo es el nuestro. Además, el proceso en el que se basan dichos algoritmos, va de la mano con temas biológicos ya expuestos en este texto. En lo que resta de esta sección, nos enfocaremos en dar una breve descripción de los algoritmos genéticos a la par de describir una implementación para el PAV. Para dar la descripción general de los algoritmos genéticos, nos basaremos el *Fundamentals of Genetic Algorithms*, de Alves da Silva y Falcão [43].

Los algoritmos genéticos se basan en simular una dinámica poblacional, donde cada individuo representa una solución factible de algún problema de optimización. Usualmente los pobladores son cadenas de números binarios, de caracteres o de números racionales. A los pobladores se les llama **cromosomas**, pues son cadenas de información. A un fragmento de información de cada cromosoma, le llamaremos **gen**. La población inicial de cromosomas puede ser generada de manera aleatoria o generada de manera sistemática con alguna heurística.

En nuestro caso, queremos que cada cromosoma cifre un ciclo hamiltoniano (una solución factible $y \in S(x)$). Anteriormente mostramos una codificación que es conocida como **representación por caminata**, la cual utiliza un vector que representa una permutación, según Larrañaga, Kuijpers, R. Murga y col. [44, pág. 137]. Dada $\sigma : \{1,\dots,n\} \to \{1,\dots,n\}$ una permutación de orden n, un cromosoma sería $y = (\sigma(1), \sigma(2), \dots, \sigma(n))$, y para evitar redundancias, pedimos que $\sigma(1) = 1$. Del ejemplo 2.1, el cromosoma asociado a la solución obtenida al final del capítulo 2 sería $y = (1,2,4,5,6,3)$ y las entradas (ciudades) serían los genes del cromosoma.

Tras cada iteración, un nuevo conjunto de cromosomas sustituirá a la población. Suponiendo que estamos en la n-ésima iteración, dicha población se define a partir de operadores de apareamiento, mutación y elitismo aplicados a la población en la iteración $n-1$. El tamaño de la población suele ser de 30 a 200 cromosomas [43, pág. 38].

La simulación de dicho proceso evolutivo se detendrá con un criterio de paro. Usualmente se escoge el número de generaciones o el tiempo de ejecución del algoritmo para detenerlo y regresar a la mejor solución encontrada.

Recordemos que los algoritmos genéticos simulan el proceso evolutivo de una población dinámica. Es importante saber qué cromosomas son los más aptos en una generación. Esto lo medimos por medio de una **función de aptitud**. Dicha función se puede elaborar a partir de la función objetivo y añadiendo penalizaciones en caso de romper la factibilidad. Mientras la aptitud de un individuo sea mayor, mayor será su probabilidad de reproducción.

Afortunadamente, la representación por caminata siempre lleva a soluciones factibles, entonces sólo debemos considerar a la función objetivo para definir a la función de aptitudes. Una forma inmediata de definirla es que dado un cromosoma de la instancia $x \in X$, la función de aptitudes para un cromosoma que represente a la solución y sería el recíproco de la función objetivo, es decir, $A(y) = 1/z(x,y)$, suponiendo que la función objetivo es positiva para toda solución factible. Recordemos que la dirección de optimización es Min, entonces mientras menor sea el valor de la función objetivo, mayor es la aptitud de un individuo.

Recordemos que los individuos deben ser apareados para proporcionar nuevos cromosomas. Ahora, las soluciones factibles deben pasar por un proceso de **selección** para saber a cuales utilizaremos para aparear. Existen diversos métodos para seleccionar a los individuos. La mayoría de ellos involucran a la función de aptitud y aleatoriedad.

Utilizaremos un método llamado **selección proporcional**. A partir de las aptitudes de los individuos, crearemos una función de distribución acumulada. Suponiendo que contamos con k cromosomas indexados y definiendo a $\overline{A}(m) = \sum_{i=1}^{m} A(i)$ para $m \in \{1,\dots,k\}$, la función F queda definida como

$$F(x) = \begin{cases} 0 & \text{si } x < 1, \\ \dfrac{\overline{A}(m)}{\overline{A}(k)} & \text{si } m \le x < m+1, \text{ para } m \in \{1,\dots,k-1\}, \\ 1 & \text{si } x > k. \end{cases} \tag{4.1}$$

Suponiendo que u es un número aleatorio del intervalo $[0,1]$, si $u \in [F(m-1), F(m))$, entonces escogemos al cromosoma con índice m para el apareamiento, con $m \in \{1,\dots,k\}$ [45, pág. 7–8]. Si $u = 1$, entonces escogemos a k.

Ya con los cromosomas seleccionados, necesitamos un proceso de **apareamiento**. Esta función va a depender de la codificación de los cromosomas. La idea es que dadas dos soluciones factibles a aparear, podamos producir una o varias soluciones factibles que compartan rasgos de las soluciones padre.

Dados dos cromosomas representados como caminatas, usaremos una función llamada **cruce conservativo maximal (MPX)** [44, pág. 146–147]. Supongamos que elegimos dos índices aleatorios $i, j \in \{2,\dots,n\}$ con $i \le j$ y dos cromosomas y_1 y y_2. La descendencia de dichos cromosomas queda definida con el proceso que se muestra a

continuación:

1. tomamos desde la i-ésima hasta la j-ésima ciudad del cromosoma y_1 y la colocamos después de la ciudad 1 en el cromosoma descendiente;
2. las ciudades restantes son colocadas en el orden dentro de y_2.

Otra implementación que se puede realizar a un algoritmo genético es un proceso de **elitismo**. Dados k cromosomas, podemos conservar a m de los mejores miembros de la población actual, con $m \in \{0,\dots,k\}$. Esto es para no redescubrir buenas soluciones tras varias iteraciones. También da lugar a mayor probabilidad para la reproducción de las mejores soluciones.

Ya considerando el proceso de elitismo, podemos crear una lista de índices de cromosomas con $k-m$ elementos obtenida a partir de la selección proporcional. Dicha lista permite la repetición de índices. De ella podemos realizar un cruzamiento con un cromosoma y su sucesor y el extremo final con el extremo inicial, así obtendremos $k-m$ nuevos cromosomas.

Finalmente, hablaremos sobre el operador de **mutación**. Recordemos que dicho proceso se define como un error aleatorio que surge durante la replicación de una molécula. En este caso, dado un cromosoma, una mutación es una alteración en el orden de las ciudades. Este operador nos permitirá tener una mayor variedad en los cromosomas. Dicha mutación ocurre con una probabilidad p, la cual suele estar en el intervalo $[0.001, 0.05]$ [43, pág. 38] . Una mutación muy práctica es la **mutación por intercambio (EM)** [44, pág. 149]. Suponiendo que un cromosoma fue elegido para modificarlo, se obtienen dos índices de forma aleatoria $i,j \in \{2,\dots,n\}$ tales que $i \leq j$, entonces se intercambian las ciudades con dicho índice.

Ejemplo 4.5. Apliquemos los algoritmos genéticos al ejemplo 2.1. Consideremos una población de 6 cromosomas, un elitismo de 2 pobladores, una probabilidad de mutación de 0.03 y una generación. Durante toda la ejecución, utilizaremos 6 cifras significativas. La población inicial es la siguiente (el superíndice señala la generación):

$$y_1^{(0)} = (1,2,3,4,5,6), \qquad y_2^{(0)} = (1,2,5,3,4,6),$$
$$y_3^{(0)} = (1,4,6,2,3,5), \qquad y_4^{(0)} = (1,6,3,4,2,5),$$
$$y_5^{(0)} = (1,5,2,3,6,4), \qquad y_6^{(0)} = (1,6,2,5,4,3).$$

Al evaluar las soluciones en la función objetivo obtenemos los siguientes valores: $z_1^{(0)} = 665$, $z_2^{(0)} = 738$, $z_3^{(0)} = 744$, $z_4^{(0)} = 834$, $z_5^{(0)} = 766$ y $z_6^{(0)} = 742$. Las aptitudes serían los recíprocos de los números anteriores, es decir, $A(i) = 1/z_i$ para $i \in \{1,\dots,6\}$. Dicho lo anterior definimos a la función de aptitud $A : \{1,\dots,6\} \to \mathbb{R}$ dada por $A(1) = 1.50375 \times 10^{-3}$, $A(2) = 1.35501 \times 10^{-3}$, $A(3) = 1.34408 \times 10^{-3}$, $A(4) = 1.19904 \times 10^{-3}$, $A(5) = 1.30548 \times 10^{-3}$ y $A(6) = 1.3477 \times 10^{-3}$. De lo anterior obtenemos que $\sum_{i=1}^{6} A(i) = 8.05509 \times 10^{-3}$.

Como nuestro parámetro de elitismo es 2, sólo necesitamos obtener 4 pobladores para la próxima generación. Utilizando el generador de números pseudo-aleatorios mencionado por Pagès [45, pág. 2] en *Numerical probability* y considerando los parámetros

$\eta_0 = 110$ (semilla), $a = 13$, $b = 17$ y $N = 907$, obtenemos los números pseudo-aleatorios (a 6 cifras significativas) $\xi_1 = 0.595369$, $\xi_2 = 0.758544$, $\xi_3 = 0.879823$ y $\xi_4 = 0.456449$. Si definimos a F a partir de A como se muestra en la ecuación 4.1, obtenemos que en los puntos $\{1,\ldots,6\}$, la función presenta saltos. La imagen de esos saltos es el conjunto $\{0.186684, 0.354902, 0.521764, 0.670619, 0.832688, 1\}$. Como $\xi_1 \in [0.521764, 0.670619)$, $\xi_2 \in [0.670619, 0.832688)$, $\xi_3 \in [0.832688, 1]$ y $\xi_4 \in [0.354902, 0.521764)$, podemos concluir que la lista de cromosomas seleccionados es $(4,5,6,3)$.

De la lista anterior, obtenemos que los apareamientos serán dados por los pares $(4,5)$, $(5,6)$, $(6,3)$ y $(3,4)$. Dicho esto, empezamos con el procedimiento de cruza. Para esta parte, necesitamos 6 números aleatorios y consideremos a los puntos equiprobables $\{2,\ldots,6\}$. Los siguientes números pseudo-aleatorios son $\xi_5 = 0.95259$, $\xi_6 = 0.402425$, $\xi_7 = 0.250275$, $\xi_8 = 0.272326$, $\xi_9 = 0.558985$, $\xi_{10} = 0.285556$, $\xi_{11} = 0.730981$ y $\xi_{12} = 0.521499$. Lo anterior nos indica que debemos tomar a los índices $(6,4)$, $(3,3)$, $(4,3)$ y $(5,4)$.

1. Del primer cromosoma de cada par ordenado y con ayuda de los índices, obtenemos una caminata parcial

$$y_1^{(1)} = (1,4,2,5,\#,\#), \qquad y_2^{(1)} = (1,2,\#,\#,\#,\#),$$
$$y_3^{(1)} = (1,2,5,\#,\#,\#), \qquad y_4^{(1)} = (1,5,4,\#,\#,\#).$$

2. Completamos los cromosomas con las ciudades restantes en el orden del segundo cromosoma de los pares ordenados.

$$y_1^{(1)} = (1,4,2,5,3,6), \qquad y_2^{(1)} = (1,2,6,5,4,3),$$
$$y_3^{(1)} = (1,2,5,4,6,3), \qquad y_4^{(1)} = (1,5,4,6,3,2).$$

Por el proceso de elitismo, definimos los últimos dos cromosomas como $y_5^{(1)} = (1,2,3,4,5,6)$ y $y_6^{(1)} = (1,2,5,3,4,6)$. Únicamente falta aplicar el operador de mutación, para esto necesitamos 6 números aleatorios. Los siguientes números pseudo-aleatorios son $\xi_{13} = 0.798235$, $\xi_{14} = 0.39581$, $\xi_{15} = 0.164277$, $\xi_{16} = 0.154355$, $\xi_{17} = 0.025358$ y $\xi_{18} = 0.348401$. Esto indica que aplicaremos una mutación al cromosoma $y_5^{(1)}$, ya que $\xi_{17} < 0.03$. Necesitamos 2 números aleatorios y consideremos a los puntos equiprobables $\{2,\ldots,6\}$. Obtenemos que $\xi_{19} = 0.54796$ y $\xi_{20} = 0.142227$, por lo que elegimos a los índices 3 y 2 y redefinimos a $y_5^{(1)} = (1,3,2,4,5,6)$.

Finalmente obtenemos que los valores del vector $z^{(1)}$, que son $z_1^{(1)} = 794$, $z_2^{(1)} = 633$, $z_3^{(1)} = 652$, $z_4^{(1)} = 678$, $z_5^{(1)} = 659$ y $z_6^{(1)} = 738$. Observemos que el promedio del vector $z^{(0)}$ es $748.1\overline{6}$ y el del vector $z^{(1)}$ es $692.\overline{3}$, lo cual nos indica que hubo una mejora en la población. Según el algoritmo genético, la ruta que deben seguir los pilotos es Ciudad de México, Guadalajara, Hermosillo, Tijuana, Los Cabos, Monterrey y regresar a Ciudad de México, con un tiempo promedio de 633 minutos. Comparado con el valor óptimo encontrado con el algoritmo de ramificación y acotamiento, sólo hay 10 minutos de diferencia.

Capítulo 5

Aplicación al fago λ

En los capítulos anteriores, estudiamos el problema del agente viajero y cómo obtener una secuencia de ADN por medio de una instancia del PAV. En este capítulo, aplicaremos dichas técnicas para secuenciar el genoma del fago λ, el cual es un virus que infecta a la bacteria E. Coli [2, pág. 194]. Como hay que realizar varios cálculos, debemos utilizar una computadora para realizar el ordenamiento. Varios de los algoritmos mencionados a lo largo de la tesis fueron implementados durante la elaboración de este trabajo en el lenguaje `Python3` [46] y pueden ser encontrados en github.com/re-vez/TesisLic. A lo largo de este capítulo, veremos cómo generar lecturas a computadora a partir de un genoma de referencia y aplicaremos la metodología descrita para ordenar a un ejemplo con el genoma mencionado.

5.1. Generación de datos *in silico* y la matriz de costos

Para aplicar nuestra metodología a ejemplos de escala real, es necesario que contemos con datos de laboratorio. Comentábamos en el capítulo 3 que nuestra metodología sólo consideraría datos sin errores, lo cual es técnicamente imposible de obtener de forma experimental. Una alternativa es generarlos por medio de una simulación en una computadora. Se le llaman experimentos *in silico* a los métodos computacionales utilizados para lo anterior y lo obtenido es útil para comprobar y desarrollar hipótesis biológicas, según Ekins, Mestres y Testa [47, pág. 9]. Muchos de estos experimentos son realizados a partir de modelos matemáticos y simulaciones.

Existen varios generadores de lecturas *in silico*, los cuales son analizados en el trabajo de Alosaimi, Bandiang, Biljon y col. [48]. Para fines prácticos, generaremos las lecturas con el proceso presentado a continuación. También desarrollaremos los supuestos que consideraremos para su obtención. A partir de los datos *in silico*, realizaremos el tratamiento mencionado en el capítulo 3. Ya con los datos tratados obtendremos la matriz de costos por medio de la ecuación 3.1. Dicha matriz representa una instancia del PAV, la cual podremos resolver. El siguiente diagrama de la figura 5.1 muestra los pasos de este procedimiento.

También en la figura 5.1, aparecen dos formatos que no hemos mencionado. El formato `.fasta` es usado para almacenar genomas secuenciados [23, pág. 202]. El formato `.npy` almacena objetos de la librería `NumPy`, la cual es una paquetería de cálculo numéri-

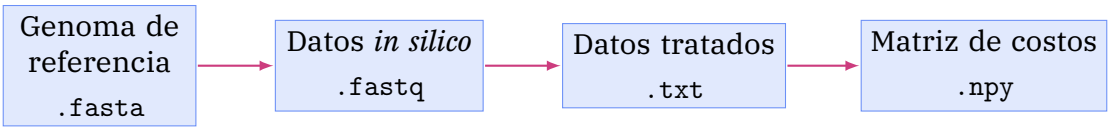


Figura 5.1: Generación de la matriz de costos.

co [49]. Ya con la matriz de costos y los datos tratados, podemos utilizar algunos de los algoritmos mencionados en el capítulo 2 o 4 para obtener la secuencia del genoma.

5.1.1. Creación del archivo FASTQ

Para la generación de las lecturas requerimos un genoma secuenciado y a partir de él se aplicaran los procesos de simulación descritos más adelante. Una base de datos genómicos de libre acceso es la del Centro Nacional para la Información de Biotecnología (o por sus siglas en inglés, NCBI), en particular el genoma que utilizaremos puede ser encontrado con el número de acceso J02459 [50]. Este genoma consta de 48502 pares de bases.

Primero necesitamos simular la duplicación del ADN. Recordemos que esto es posible por medio de la PCR y teniendo los primers adecuados que se unen a la molécula. **Supuesto 1:** En el medio del experimento se encuentran únicamente la molécula a secuenciar y todos los materiales necesarios para hacer una cantidad arbitraria de copias de la secuencia. Si realizamos C ciclos en la PCR con una única copia de ADN en el medio, en teoría obtendremos 2^C réplicas.

Muchos de los organismos conocidos tienen ADN de doble hélice, por lo que es necesario separar a la cadena codificante de la cadena complementaria. Romper los puentes de hidrógeno que unen a las hélices es posible al elevar la temperatura en el medio. **Supuesto 2:** Somos capaces de retirar a las cadenas complementarias del medio. De esta forma, no tendremos que realizar un análisis de datos exhaustivo para identificar a qué hebra corresponde cada una de las lecturas que obtendremos más adelante.

Gusfield [1, pág. 160] comenta que en la secuenciación por escopeta, los cortes para obtener las lecturas deben ser aleatorios. Mencionamos anteriormente que podemos romper las moléculas de ADN de manera física, lo cual resulta ser un proceso aleatorio. **Supuesto 3:** El evento aleatorio donde se rompe un enlace fosfodiéster en particular, es independiente al rompimiento de los demás y todos estos eventos tienen la misma probabilidad, la cual podemos manipular. El hecho de que todos los enlaces tengan la misma probabilidad de romperse, reduce la cantidad de cálculos a realizar.

Entrando más a detalle en el proceso aleatorio de los cortes. Dada $|\lambda|$ la longitud del genoma, necesitamos simular un conjunto de variables aleatorias independientes e idénticamente distribuidas $\{X_i\}_{i=1}^{n(2^C)+1}$ con $n : \mathbb{N} \to \mathbb{N}$ una función estrictamente creciente tal que $n(0) = 0$ y

$$\sum_{j=n(i-1)+1}^{n(i)} X_j \leq |\lambda| < \sum_{j=n(i-1)+1}^{n(i)+1} X_j, \quad \text{para toda } i \in \left\{1, \ldots, 2^C\right\}. \tag{5.1}$$

En la figura 5.2 se encuentra ilustrada la condición (5.1). El número $\sum_{j=n(i-1)+1}^{n(i)-k} X_j$ nos

indicará un sitio de corte en la i-ésima copia, con $k \in \{1,\ldots,n(i)-n(i-1)\}$ e $i \in \{1,\ldots,2^C\}$. Las variables aleatorias $X_i's$ son tales que $X_i - 1 \sim \text{Geo}(p)$. La esperanza de dicha variable aleatoria es $1/p$, por lo que si queremos que los fragmentos tengan en promedio una longitud ℓ, definimos a $p = 1/\ell$.

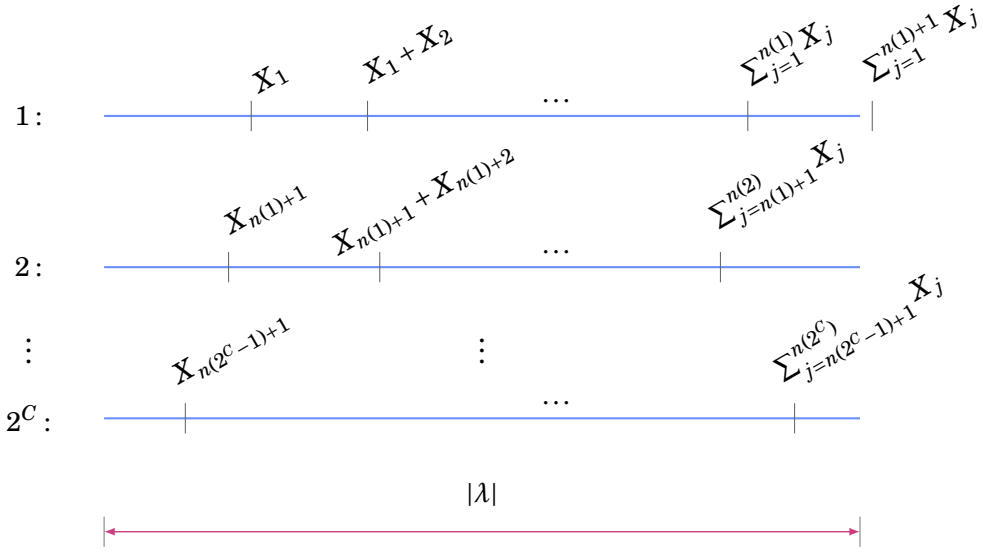


Figura 5.2: Cortes aleatorios.

Como resultado obtendremos un conjunto de moléculas de ADN que podemos secuenciar en una máquina *MiSeq*. **Supuesto 4:** No hay errores durante la secuenciación de las lecturas. Otro parámetro a considerar son los ciclos para determinar las bases leídas de cada molécula; dependiendo del tipo de secuenciador que utilizado, puede variar la cantidad de letras observadas [51]. Dados un fragmento de longitud l y c los ciclos que realiza la máquina, el número de bases secuenciadas será $\text{mín}\{c,l\}$. Como no nos interesa la calidad, pues son datos generados sin errores, podemos incluir una calidad simbólica como un parámetro. Tras esto, podemos obtener un archivo `.fastq` que contenga a las lecturas del proceso anterior. El código que generador de lecturas se llama `Lectures.py`.

Las variables aleatorias de nuestro ejemplo fueron generadas con la paquetería `NumPy`. Al aplicar el proceso anterior al genoma del fago λ se obtuvieron 2558 lecturas. Los parámetros utilizados fueron: 200 ciclos en el secuenciador MiSeq, 4 ciclos de replicación de PCR del genoma, una calidad de 40 unidades, un promedio de corte en cada 300 pares de bases y la semilla 13013 (para la generación de números pseudo-aleatorios con `NumPy`). El resultado anterior puede ser consultados en el archivo `J02459.fastq`.

5.1.2. Limpieza de datos y matriz de costos

Comentamos en el capítulo 3 que necesitamos verificar que ninguna cadena en el archivo `.fastq` sea subcadena de otra cadena del mismo archivo, para poder definir a los costos sin problema alguno. Primero debemos eliminar las lecturas que estén contenidas en otras. Por un lado, mencionamos que un método para hacerlo es utilizando el algoritmo de Boyer y Moore [27]. Por otro lado, `Python3` tiene implementada la función

`find()` que dadas dos cadenas regresa el índice donde comienza la subcadena, si se encuentra, o -1 en otro caso. Dicho esto, nos auxiliaremos de la función implementada.

Recordemos que el tratamiento de datos consiste en dos pasos. Primero necesitamos considerar una longitud mínima ℓ para las lecturas lo cual nos dará un conjunto Λ. Esto es para desechar datos que aporten poca evidencia para la secuenciación. Una vez descartadas las lecturas cortas necesitamos verificar que no existan pares de ellas $(u,v) \in \Lambda \times \Lambda$ tales que u sea subcadena de v, en caso de que esto ocurra, redefinimos a Λ como $\Lambda \backslash \{u\}$ hasta eliminar todas las cadenas que satisfagan lo anterior. El código que realiza el tratamiento de las lecturas se encuentra en el archivo `NoErrorTreatment.py`.

Al aplicar el procedimiento a las lecturas del archivo `J02459.fastq`, el conjunto se redujo a 1466 lecturas. La longitud mínima que pedimos fue de 50 bases. Los resultados de este procedimiento pueden ser consultados en el archivo `Treated.txt`, donde cada línea de código representa una lectura. La suma de las longitudes de todas las lecturas es de 339491, entonces tenemos una cobertura general de $339491/48502 \approx 6.99952$.

En el capítulo 4 comentamos que una forma de representar instancias del PAV es por medio de matrices. Si la instancia cuenta con n ciudades, necesitamos una matriz C de $n \times n$ entradas. Cada entrada $c_{i,j}$ de dicha matriz estará dada por la ecuación (3.1) para $i,j \in \{1,\ldots,n\}$, en caso de que $i \neq j$ y $c_{i,i} = \infty$. Recordemos que en esta matriz debemos resolver un problema del mensajero, por lo que debemos extender la matriz en una fila y una columna al final. Con lo anterior, tenemos una extensión de C de tamaño $(n+1)\times(n+1)$ tal que $c_{i,n+1} = 0$, $c_{n+1,j} = 0$ y $c_{n+1,n+1} = \infty$. En dicha matriz extendida resolveremos el PAV para obtener el ordenamiento de las lecturas. El código del procedimiento anterior se encuentra en los archivos `OverlapMatrix.py` y `OverlapMatrixNew.py`.

5.2. Secuenciación del genoma *Escherichia phage Lambda*

En esta etapa, ya podemos aplicar los algoritmos de optimización descritos para obtener el ordenamiento de las lecturas. Recordemos que dichos métodos son el algoritmo de ramificación y acotamiento y el algoritmo genético. Ya con el ordenamiento, sólo resta acomodar a las lecturas para obtener la secuencia del genoma. Para verificar qué tan parecidos son los genomas, utilizaremos la distancia de Levenshtein entre las dos cadenas dividida por el máximo entre la longitud de ambas.

Notemos que en nuestro ejemplo, la cobertura general de las bases es de 7, lo cual es muy reducido a lo recomendado. La razón por la que trabajaremos con estos datos es para utilizar una instancia de tamaño manejable del problema. En caso de no hacerlo, el único algoritmo que podríamos ejecutar sería el algoritmo genético. El algoritmo de ramificación y acotamiento podría no terminar ya que utilizaría toda la memoria RAM de la computadora donde se ejecute.

5.2.1. Algoritmo de ramificación y acotamiento

Para obtener la secuenciación del genoma a partir de la matriz de costos, el diagrama ilustrado en la figura 5.3 resume las transformaciones a realizar. La matriz de dimensión

$(n+1)\times(n+1)$ será la entrada del algoritmo. Dicho algoritmo se encuentra implementado en el archivo `Cap.py` con el nombre `BnB`. Éste nos regresará una matriz X de tamaño $(n+1)\times(n+1)$, donde $x_{i,j}=1$ si se va de la ciudad i a la ciudad j en otro caso.

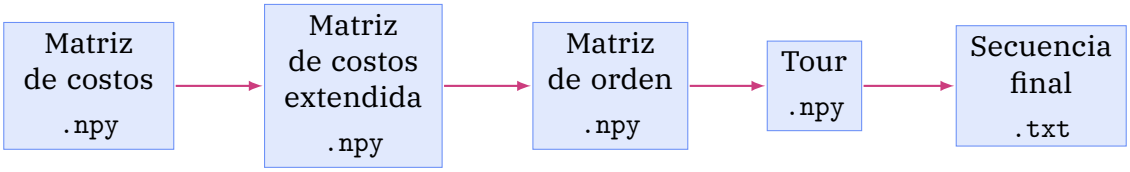


Figura 5.3: Obtención de la secuencia con el algoritmo de RyA.

Ya con la matriz X, necesitamos obtener la caminata para realizar el ordenamiento. El camino hamiltoniano será almacenado en un vector t de dimensión n de `NumPy` por medio de una representación de caminata y utilizaremos un contador $c=1$. Para esto, nos fijaremos en la j tal que $x_{n+1,j}=1$, donde definiremos a $t(1):=j$ y $c:=2$. A partir de aquí, definiremos a $i:=j$ y redefiniremos a j como la entrada tal que $x_{i,j}=1$, a $t(c):=j$ y finalmente a $c:=c+1$. El proceso anterior se repetirá hasta que c sea igual a n. El código de este procedimiento se encuentra en el archivo `MatrixToPath.py`.

Tenemos una matriz C de tamaño 1467×1467. La cota inicial del algoritmo fue $w=98702$ y el valor óptimo $z^*=98703$, el cual se obtuvo después de examinar 2951 nodos. Este procedimiento utilizó de aproximadamente 46 GB de RAM, dato a considerar para (no) replicar el proceso. Dicho código se encuentra en el archivo `Order.py`.

Al ordenar las lecturas, obtuvimos que la secuencia tiene una longitud de 49551 bases y tiene una distancia de Levenshtein de 7115 bases. Esto nos indica que el genoma original y nuestro genoma secuenciado se parecen en un 85.643074%. Este resultado se encuentra en el archivo `Escherichia_phage_Lambda_seq_4.txt` y fue generado por el código del archivo `FinalDNA.py`. Las primeras 35513 bases son idénticas entre los dos genomas. En el camino hamiltoniano se encuentra el arco que une a las lecturas 814 y 1434 de costo 311 (la suma de la longitud de las lecturas), las cuales son cortas y no tienen traslapes. Es posible que esto haya sido ocasionado por la falta de cobertura.

5.2.2. Algoritmo genético

En el capítulo 4 se utilizó un algoritmo genético para el PAV, pero podemos hacerle unas pequeñas modificaciones para que este resuelva instancias del PM.

- Omitiremos la condición de que $\sigma(1)=1$, pues no podemos fijar una ciudad inicial.
- El costo de la caminata $y=\langle\sigma\rangle$ estará dado por $c(y)=\sum_{i=1}^{n-1} c_{\sigma(i),\sigma(i+1)}$.
- Los números aleatorios para la elección de los índices en el cruce conservativo maximal serán tomados del conjunto $\{1,\ldots,n\}$, al igual que los índices de mutación.

Con las modificaciones anteriores, no es necesario extender a la matriz C obtenida al ejecutar el código del archivo `OverlapMatrix.py`. Para obtener la secuenciación del genoma a partir de la matriz de costos, el diagrama ilustrado en la figura 5.4 resume los archivos a ejecutar.

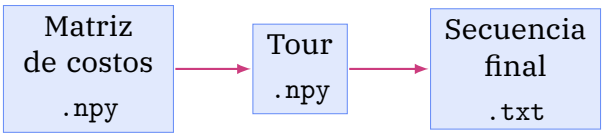


Figura 5.4: Obtención de la secuencia con el algoritmo genético.

En el archivo `OrderGen.py` se encuentra el código necesario para obtener un camino hamiltoniano que aproxime al óptimo de la instancia dada por la matriz C. El algoritmo es una función del archivo `Cap4.py` y tiene el nombre de `geneticAlgorithm`. Hay otra versión que guarda los mejores caminos de cada generación para posteriormente ser graficados, dicha función tiene el nombre de `geneticAlgorithmPlot`. Los parámetros que se utilizaron para el algoritmo son 5000 generaciones, 200 pobladores, 10 pobladores de élite y 0.05 % de probabilidad de mutación. Los parámetros de población y mutación fueron seleccionados como los más grandes sugeridos por Alves da Silva y Falcão [43, pág. 38] y el parámetro de generaciones fue seleccionado en función del tiempo ejecución del algoritmo anterior. Se obtuvo una solución y con valor $z(y) = 522208$, la cual fue encontrada en la generación 3231. En la figura 5.5 se muestra el desempeño del algoritmo. Notemos que el error absoluto con respecto al valor óptimo es 423505 y el error relativo es 4.2907.

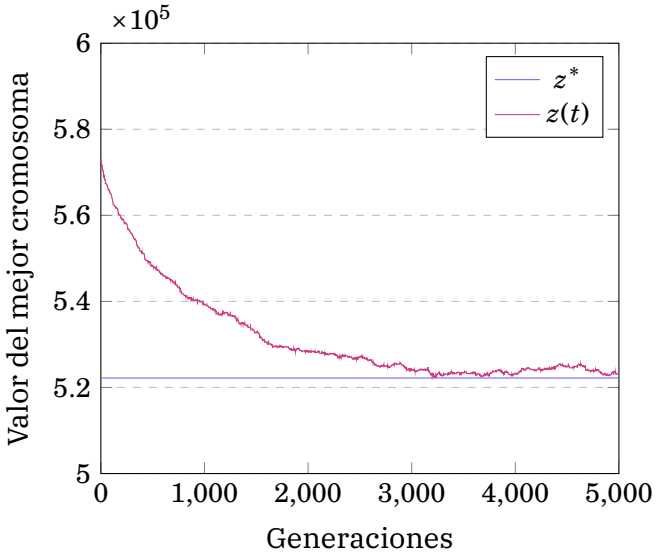


Figura 5.5: La mejor ejecución del algoritmo genético.

El genoma obtenido por medio de la solución anterior, tiene una longitud de 261304 bases. No fue posible comparar las cadenas para saber en qué proporción se parecen, ya que para obtener la distancia de Levenshtein, se necesitan 94.4 GB de memoria RAM (la memoria necesaria para generar un arreglo matricial de `NumPy` de 48503×261305 entradas). Por la cota inferior de la distancia de Levenshtein, podemos asegurar que las secuencias se parecen a lo más en un 18.5619 %. Dicho genoma se encuentra en el archivo `Escherichia_phage_Lambda_seq_Gen.txt`.

Conclusión

A lo largo de esta tesis planteamos un modelo matemático para resolver un problema de índole biológico. El ordenamiento de lecturas es utilizado para obtener un genoma de referencia, o como menciona Langmead [4], la caja del rompecabezas.

Para lograrlo, describimos un método de secuenciación de segunda generación, estudiamos un método exacto y un heurístico para el problema de optimización conocido como del agente viajero, el cual utilizamos para obtener dicho ordenamiento. Adicional a lo anterior, evidenciamos que el PAV es un problema difícil de resolver por medio de la teoría de complejidad computacional.

Debido a que el problema es NP–Duro, a mayor cantidad de lecturas a ordenar, más tardada será la ejecución de los algoritmos conocidos para resolver el problema y posiblemente no se puedan ejecutar varios a falta de tiempo o de memoria, como es el caso del algoritmo de ramificación y acotamiento. Si P = NP entonces existiría algún algoritmo eficiente para resolver el PAV, pero hay mucho escepticismo y posiblemente P esté estrictamente contenido en NP.

A causa de lo anterior, las heurísticas resultan ser una alternativa útil en la práctica, como lo presentan Kato y Hasegawa [52] con un algoritmo para el modelo presentado en el capítulo 3. El algoritmo genético presentado para el PM no resultó dar buenas aproximaciones en poco tiempo, esto no debe ser motivo para demeritar los métodos heurísticos. Una modificación que se puede realizar es aplicar un método glotón con algunas perturbaciones aleatorias para la generación de soluciones iniciales del algoritmo genético. Lo anterior es justificado por Ma [53].

Cabe mencionar que la metodología para el ordenamiento de fragmentos, desarrollada a lo largo de este trabajo, no es la única que podemos aplicar. Existe una desventaja importante al utilizar las lecturas de un secuenciador *MiSeq* en conjunto con el PAV, debido a que son demasiado cortas y esto puede generar dificultades al momento de tener k-meros repetitivos en una secuencia. Una manera de ejemplificar lo anterior es a través de los llamados telómeros, secuencias que en los humanos son constituidas por 6-meros de la forma `TTAGGG`, mismos que se repiten alrededor de 2500 veces en la cadena codificante de los extremos de cada cromosoma [2, pág. 182]. Esto es mucho mayor a la longitud máxima de un secuenciador de *Illumina*.

Para aventajar el inconveniente anterior, existen nuevas tecnologías de secuenciación donde la longitud de las lecturas suele exceder las diez mil bases [3, pág. 14], pero presentan tasas de error altas (entre 5 y 15 %) en comparación con las de *Illumina* [28];

dichas fallas pueden ser corregidas con ayuda de la cobertura, la cual es definida antes del ejemplo 3.7.

Una metodología utilizada para las secuencias de una MiSeq según Simpson [28], es la propuesta por Pevzner, Tang y Waterman [54] que consiste en construir gráficas de De Bruijn, en donde los caminos eulerianos nos indican cómo ensamblar las lecturas para obtener la secuencia original. A diferencia de nuestro procedimiento, aquí se busca verificar la existencia de dicha caminata o encontrar lo más cercano a una. Aunque las gráficas generadas son de gran tamaño, comprobar lo anterior se puede efectuar en tiempo polinomial [55].

Aunado a lo anterior, no necesariamente tendremos la certeza suficiente para decir que dos lecturas son sucesivas, por ende, las lecturas suelen utilizarse para ensamblar supercadenas llamadas **cóntigos**. Con estas estructuras y utilizando técnicas de bioinformática, se puede obtener un genoma. Este procedimiento es usado por secuenciadores como *ARACHNE*, el cual fue desarrollado por Batzoglou, Jaffe, Stanley y col. [56].

Para este trabajo elegimos la tecnología de segunda generación con el fin de utilizar datos experimentales tratados. Finalmente, esto no se realizó porque implicaba el desarrollo de subrutinas adicionales para obtener el genoma con lecturas ordenadas a partir de los traslapes aproximados, lo cual no era el objetivo del trabajo. Como mencionamos en el capítulo 1, otra razón para la elección de esta tecnología es su bajo costo.

Ya con el genoma de referencia, será más sencillo obtener la secuencias de otro individuo de la misma especie. Usualmente la referencia y la molécula de ADN que buscamos secuenciar suelen coincidir en más del 99 %. Aquí se emplean métodos como el algoritmo de Boyer y Moore para aparear las lecturas con algún k-mero del genoma y así armar toda la nueva secuencia. Un algoritmo para este tipo de secuenciación es el de Nsira, Lecroq y Elloumi [57].

Por último, conocer las secuencias de ADN es fundamental para el área de epigenética, la cual se dedica al estudio de los genes, así como su expresión y sus modificaciones [58]. Al conocer su ubicación en el genoma y su proteína asociada, es posible saber cómo se efectúan las modificaciones, tanto al código genético como a su ambiente, para favorecer o transformar su expresión. Esta rama es fundamental en los estudios de salud, pues diversos de ellos han comprobado que dichas alteraciones desembocan en enfermedades como el cáncer, la diabetes, entre otras. En el 2005, se fundó un proyecto que se dedica al mapeo de ellas, llamado *el proyecto del epigenoma humano*. Afortunadamente, hay artículos que demuestran que dichos procesos pueden ser revertidos por medio de fármacos, aunque será con el tiempo y los avances científicos que veremos si es posible prevenir o curar estas patologías.

El trabajo en síntesis:

Llegando a los orígenes de la vida por el camino de menor costo.

Apéndice A

Elementos de Investigación de Operaciones

En este apéndice se presentan elementos básicos de ramas de la investigación de operaciones, como lo son de gráficas, redes y programación matemática. Los conceptos expuestos a continuación, son necesarios para comprender los capítulos 2, 3 y 4 del trabajo.

A.1. Gráficas y redes

Esta sección del apéndice A fue elaborada a partir de los libros *Introducción a la teoría de redes* de Hernández–Ayuso [17, ap. A] y *Assignment problems* de Bukard, Dell'Amico y Martello [11, cap. 3].

Definición A.1. Definimos una **gráfica** G como la pareja de conjuntos $G = (X, A)$, donde X es un conjunto no vacío llamado el conjunto de **nodos** y $A \subseteq X \times X$ llamado el conjunto de **arcos** en caso de tener direcciones o **aristas** no tenerlas. Nótese que para cada elemento $a \in A$, existen $i, j \in X$ tales que se puede representar a a como $a = (i, j)$, donde a i se le denota el **nodo o extremo inicial de** a y a j se le denota como el **nodo o extremo final de** a. En este texto, supondremos que X es un conjunto finito.

Definición A.2. Dados dos arcos (o aristas), decimos que son **arcos adyacentes** (o **aristas adyacentes**) si tienen extremos en común.

Definición A.3. Sea $G = (X, A)$ una gráfica (dirigida o no dirigida). Definimos a una **cadena** como la secuencia de aristas $a_1, ..., a_q$ en la cual a_i comparte un extremo con a_{i+1}, para toda $i \in \{1, ..., q-1\}$. Notemos que dos arcos sucesivos en la cadena son adyacentes. En caso de tener una gráfica dirigida, decimos que la cadena $a_1, ..., a_q$ es un **camino** si el extremo final de a_i es igual al extremo inicial de a_{i+1}, para toda $i \in \{1, ..., q-1\}$.

Definición A.4. Una **cadena elemental (o camino elemental)** es una cadena (o camino) tal que no repite nodos, i.e., dadas $a_p, a_q \in A$ aristas (o arcos) tales que $a_p = (i_1, j_1)$ y $a_q = (i_2, j_2)$ con $p \neq q$, entonces $i_1 \notin \{i_2, j_2\}$ y $j_1 \notin \{i_2, j_2\}$.

Definición A.5. Un **ciclo** es una cadena $a_1, a_2, ..., a_q$ tal que a_1 y a_q tienen un extremo en común. Un **circuito** es un ciclo $a_1, a_2, ..., a_q$ tal que el extremo inicial de a_1 coincide con el extremo final de a_q.

Definición A.6. Un **circuito (o ciclo) hamiltoniano** es un circuito (o ciclo) elemental que contiene todos los nodos de la gráfica. Una **cadena (o camino) hamiltoniano** es una cadena (o camino) elemental que contiene todos los nodos de la gráfica.

Definición A.7. Un **circuito (o ciclo) euleriano** es un circuito (o ciclo) que contiene todas las aristas (o los arcos) de la gráfica sin repetirlas. Una **cadena (o camino) euleriano** es una cadena (o camino) que contiene todas las aristas (o los arcos) de la gráfica sin reperitlas.

Definición A.8. Sean $i, j \in X$. Decimos que j es **vecino** de i si $(i,j) \in A$ o $(j,i) \in A$. Si G es una digráfica, decimos que j es **predecesor** de i si $(j,i) \in A$ o que j es **sucesor** de i si $(i,j) \in A$. Definimos a las funciones $\Gamma^+, \Gamma^-, \Gamma : X \to \mathscr{P}(X)$ dadas por $\Gamma^+(i) = \{j \in X : (i,j) \in A\}$, $\Gamma^-(i) = \{j \in X : (j,i) \in A\}$ y $\Gamma(i) = \Gamma^+(i) \cup \Gamma^-(i)$.

Definición A.9. Sea $X' \subseteq X$, definimos una **subgráfica de** G como $G' = (X', A')$ donde $A' = \{(i,j) \in A : i, j \in X'\}$. Definimos a una **gráfica parcial de** G como $G' = (X, A')$ donde $A' \subseteq A$.

Definición A.10. Sea $G = (X, A)$ una gráfica. Decimos que G es una gráfica **conexa** si para cualquier par de nodos, existe una cadena que los una. Si $X_1, \ldots, X_k$ es una partición de X donde cada (X_i, A_i) es una subgráfica gráfica conexa y para cada $x \in X_i$ y $y \in X_j$ no existe una cadena que los una para toda $i, j \in \{1, \ldots, k\}$ con $i \neq j$, decimos que $X_1, \ldots, X_k$ son las **componentes conexas** de G.

Definición A.11. Sea $G = (X, A)$ una gráfica y $d \in \mathbb{Z}^+$. Una **red** R es una terna formada por $R = (X, A, f)$, donde f es una función con valores en $\mathbb{R}^d$ y dominio en X, A o $X \cup A$.

Definición A.12. Sea $G = (X \cup X', A)$ una gráfica donde X, X' son conjuntos ajenos. Decimos que G es una **gráfica bipartita** se cumple que para todo $(i,j) \in A$, $i \in X$ y $j \in X'$ o viceversa.

Definición A.13. Decimos que la gráfica G es un **árbol** si G es una gráfica acíclica y conexa.

Definición A.14. Sea $G = (X, A)$ una gráfica y $M \subseteq A$. Decimos que M es un **acoplamiento** de G si los arcos dentro de M no son adyacentes entre sí. Decimos que una arista $a \in A$ es una arista acoplada si $a \in M$.

Definición A.15. Sea $G = (X, A)$, $M \subseteq A$ un acoplamiento de G y $a_1, a_2, \ldots, a_q$ un camino. Decimos $a_1, a_2, \ldots, a_q$ es un **camino M-alternante** si dado un arco perteneciente a M, entonces su antecesor y su sucesor en el camino no pertenecen a M, o si dado un arco que no pertenece a M, entonces su antecesor y su sucesor en el camino sí pertenecen a M.

Definición A.16. Sea $G = (X, A)$, $M \subseteq A$ un acoplamiento de G y $a_1, a_2, \ldots, a_q$ un M-camino alternante tal que el extremo inicial de a_1 está en X y el extremo final de a_q está en X'. Decimos $a_1, a_2, \ldots, a_q$ es un **camino M-aumentante** si hay aristas conectadas a dichos extremos que no sean pertenecientes a M.

Definición A.17. Sea $G = (X, A)$, $T = (N, A')$ un árbol con $N \subseteq X$ y $A' \subseteq A$, $M \subseteq A$ un acoplamiento de G, y $k \in N$. Decimos que T es un **árbol M-alternante enraizado en k** si todos los caminos que se puedan definir en T con nodo inicial k son M-alternantes y todos los caminos maximales (los que van a nodos terminales del árbol) desde la raíz son de cardinalidad par.

A.2. Programación matemática

Esta sección del apéndice A fue elaborada a partir de los libros *Linear Programming* de Karloff [59, cap. 1 y 3], *Integer Programming* de Conforti, Cornuejols y Zambelli [12, pág. 1–5], *Linear Algebra and Geometry* de Shafarevich y Remizov [60] y *Numerical Lineal Algebra* de Allaire y Kaber [15, pág. 19].

Definición A.18. Dado $n \in \mathbb{Z}^+$, definimos a una **permutación de orden** n como una función inyectiva $\sigma : \{1,\dots,n\} \to \{1,\dots,n\}$. Definimos a $\mathscr{S}_n$ como el **conjunto de todas las permutaciones de orden** n.

Observación A.19. Recordemos que los ordenamientos de orden n pueden ser representados con vectores de $\mathbb{R}^n$.

Teorema A.20 (de Rouché–Capelli)**.** *Dado un sistema de ecuaciones en forma matricial $Ax = b$ donde $A \in M_{n\times m}(\mathbb{R})$, $x \in \mathbb{R}^n$ y $b \in \mathbb{R}^m$, entonces el sistema tiene solución si y sólo si el rango de A coincide con el rango de la matriz extendida $[A|b]$.*

Definición A.21. Sea $F \subseteq \mathbb{R}^n$, decimos que F es un **conjunto convexo** si para cualesquiera $x, y \in F$ y $\lambda \in [0,1]$, entonces $\lambda x + (1-\lambda)y \in F$. Decimos que $z \in F$ es un **punto extremo** de F si no existen $x, y \in F$ distintos y $\lambda \in (0,1)$ tal que $z = \lambda x + (1-\lambda)y$.

Observación A.22. Dados $x, y \in \mathbb{R}^n$ con $x = (x_1,\dots,x_n)^t$ y $y = (y_1,\dots,y_n)^t$, decimos que $x \geq y$ si $x_i \geq y_i$, para toda $i \in \{1,\dots,n\}$. Decimos que x es **entero** si cada entrada de x es entera.

Definición A.23. Definimos un **problema de programación lineal (o modelo lineal) en forma canónica** como

$$\begin{array}{ll} \text{Max} & c^t x, \\ \text{s.a} & Ax \leq b, \\ & x \geq 0, \end{array}$$

donde $c, x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ y $A \in M_{n\times m}(\mathbb{R})$ con A de rango m. Si en lugar de la desigualdad $Ax \leq b$ tenemos $Ax = b$, entonces el problema lineal está en **forma estándar**. Definimos a un **problema de programación lineal entera (o modelo lineal entero)** como

$$\begin{array}{ll} \text{Max} & c^t x, \\ \text{s.a} & Ax \leq b, \\ & x \geq 0, \\ & x \text{ entero}. \end{array}$$

En caso de que sólo necesitemos que algunas entradas de x sean enteras, entonces el problema anterior es conocido como **programación lineal entero mixto**. En particular, si x es un vector con entradas binarias ($\{0,1\}$), decimos que el problema anterior es un problema de **programación lineal binario**. A cada ecuación, desigualdad o condición del sistema se le conoce como **restricción**. A la función $z(x) = c^t x$ se le llama **función objetivo**.

Observación A.24. Observemos que todo problema lineal en la forma estándar se puede transformar a la forma canónica si partimos a cada fila $a_i^t x = b_i$ del sistema de ecuaciones en las dos desigualdades $-a_i^t x \leq -b_i$ y $a_i^t x \leq b_i$, para $i \in \{1,\dots,m\}$. De la misma forma, las desigualdades del problema canónico se pueden transformar en ecuaciones al aumentar en n dimensiones el problema con las variables $x_k' s \geq 0$ (llamadas **variables de holgura**) para así tener m ecuaciones de la forma $a_i^t x + x_k \leq b_i$, para $k \in \{n+1\dots,n+m\}$.

Teorema A.25. *El conjunto $F = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$ es un conjunto convexo. Además, si existe $M = \max\{c^t x : x \in F\}$ con $c \in \mathbb{R}^n$, entonces existe x^* un punto extremo de F tal que $c^t x^* = M$. Lo anterior también se satisface para $F' = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$.*

Definición A.26. Consideremos el problema lineal en su forma estándar de la definición A.23. Supongamos que $m \leq n$ y A es de rango completo. Decimos que un vector $x^* \in \mathbb{R}^n$ que satisface el sistema anterior es una **solución básica factible (sbf)** si dados los índices $I = \{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\}$ y la matriz $B = [A_{j_1} | \ldots | A_{j_m}]$ cuyas columnas forman una base de $\mathbb{R}^m$, se tiene que las entradas $x^*_k = (B^{-1}b)_k$ (llamadas **variables básicas**) para $k \in I$ y las entradas $x^*_k = 0$ (llamadas **variables no básicas**) para $k \in \{1, \ldots, n\} \backslash I$.

Teorema A.27. *Consideremos el problema lineal en su forma estándar de la definición A.23. Entonces x es sbf del problema lineal si y sólo si es punto extremo de la región definida por sus restricciones.*

Definición A.28. Dado el problema de programación lineal en forma canónica de la definición A.23, definimos su **problema dual** como

$$\begin{array}{ll} \text{Min} & b^t\, w, \\ \text{s.a} & A^t w \geq c, \\ & w \geq 0. \end{array}$$

donde $w \in \mathbb{R}^m$. En este contexto, al problema lineal original se le conoce como **problema primal**.

Teorema A.29 (de dualidad débil y fuerte)**.** *Considere un problema de programación lineal y su problema dual. Sea $x \in \mathbb{R}^n$ una solución factible del primal y $w \in \mathbb{R}^m$ una solución factible del dual, entonces se satisface que $c^t x \leq b^t w$. La igualdad se da si y sólo si x y w son soluciones óptimas del primal y dual respectivamente.*

Teorema A.30 (de holguras complementarias)**.** *Considere un problema de programación lineal y su problema dual. Sea $x \in \mathbb{R}^n$ una solución factible del primal y $w \in \mathbb{R}^m$ una solución factible del dual, entonces las soluciones son óptimas si y sólo si*

$$(A^t w - c)^t x = 0 \quad y \quad (Ax - b)^t w = 0.$$

Definición A.31. Consideremos el problema entero lineal de la definición A.23. Si retiramos la restricción de que las entradas del vector x sean enteras, entonces ese problema se conoce como la **relajación PL** del problema entero. En general, decimos que un problema es la **relajación** de otro si toda restricción del primero es restricción del segundo.

Definición A.32. Sean $f, g : \mathbb{N} \to \mathbb{R}^+$ funciones. Decimos que $g(n)$ es un **límite superior asintótico** de $f(n)$ (o bien, $f(n)$ es $\mathcal{O}(g(n))$) si existen $c, n_0 \in \mathbb{Z}^+$ tales que para toda $n \geq n_0$, se satisface que $f(n) \leq cg(n)$.

Apéndice B

Pseudocódigos y diagramas de flujo

En este apéndice se presentan los pseudocódigos cada algoritmo y los diagramas de flujo de algunos. Todos los algoritmos presentados a lo largo de este trabajo fueron implementados en el lenguaje de programación `Python 3`. Los códigos se encuentran en el repositorio github.com/re-vez/TesisLic.

B.1. Capítulo 2

Algoritmo 1: Preprocesamiento.

Datos: Una matriz C de costos.
Resultado: $\overline{U}$ conjunto inicial de asignaciones,
φ un inicial vector de asignación de personas,
f un vector inicial de asignación de tareas,
(u, v) una solución dual factible.

para $i \in \{1, \ldots, n\}$ **hacer**
    $u_i := \min\{c_{i,j} : j \in \{1, \ldots, n\}\}$;
para $j \in \{1, \ldots, n\}$ **hacer**
    $v_j := \min\{c_{i,j} - u_i : i \in \{1, \ldots, n\}\}$;
para $i \in \{1, \ldots, n\}$ **hacer**
    para $j \in \{1, \ldots, n\}$ **hacer**
        si $f(j) = \infty$, $c_{i,j} - u_i - v_j = 0$ y $i \notin \overline{U}$ **entonces**
            $f(j) := i$, $\varphi(i) := j$, $\overline{U} := \overline{U} \cup \{i\}$;

Algoritmo 2: Procedimiento alternante.

Datos: k nodo expuesto de la gráfica actual,
φ un inicial vector de asignación de personas,
f un vector inicial de asignación de tareas,
(u,v) una solución dual factible.
Resultado: s un nodo expuesto mediante el cual haremos el procedimiento alternante,
pred el vector de predecesores actualizado.

$\pi(j) := \infty$, para toda $j \in \{1,...,n\}$;
$SU = SV = LV := \varnothing$, $s := \infty$, $i := k$;
mientras $s = \infty$ **hacer**
  $SU = SU \cup \{i\}$;
  para $j \in V \setminus LV$ *tales que* $\overline{c}_{i,j} < \pi(j)$ **hacer**
    $\text{pred}(j) = i$, $\pi(j) = \overline{c}_{i,j}$;
    si $\pi(j) = 0$ **entonces**
      $LV = LV \cup \{j\}$
  si $LV \setminus SV = \varnothing$ **entonces**
    $\delta = \text{mín}\{\pi(j) : j \in V \setminus LV\}$;
    para $i \in SU$ **hacer**
      $u_i := u_i + \delta$;
    para $j \in LV$ **hacer**
      $v_j := v_j - \delta$;
    para $j \in V \setminus LV$ **hacer**
      $\pi(j) := \pi(j) - \delta$;
      si $\pi(j) = 0$ **entonces**
        $LV := LV \cup \{j\}$;
  Sea $j \in LV \setminus SV$;
  $SV := SV \cup \{j\}$;
  si $f(j) = 0$ **entonces**
    $s := j$;
  en otro caso
    $i := f(j)$;

Algoritmo 3: Húngaro.

Datos: $\overline{U}$ conjunto inicial de asignaciones,
φ un inicial vector de asignación de personas,
f un vector inicial de asignación de tareas,
(u,v) una solución dual factible.
Resultado: v
mientras $\#\overline{U} < n$ **hacer**
  Sea $k \in U \setminus \overline{U}$;
  $\text{pred}(q) = k$, para toda $q \in \{1,...,n\}$;
  Sean j, pred obtenidos del procedimiento alternante aplicado con el nodo k;
  $\overline{U} := \overline{U} \cup \{k\}$;
  $i := \text{pred}(j)$, $f(j) := i$, $h := \varphi(i)$, $\varphi(i) := j$, $j := h$;
  repetir
    $i := \text{pred}(j)$, $f(j) := i$, $h := \varphi(i)$, $\varphi(i) := j$, $j := h$;
  hasta que $i = k$;

Algoritmo 4: Resultados del algoritmo húngaro.

Datos: C la matriz de costos,
$(u,v) \in \mathbb{R}^n \times \mathbb{R}^n$ la solución dual obtenida por el algoritmo húngaro,
φ el vector de asignación de personas con respecto a las tareas.
Resultado: X una matriz de asignación óptima,
z el valor óptimo de la función objetivo,
$\overline{C}$ la matriz de costos reducidos asociada a la solución obtenida.
Sea $X, \overline{C} \in M_{n\times n}(\mathbb{R})$ matrices nulas y $z := 0$;
para $i \in \{1,\dots,n\}$ **hacer**
  $z := z + u_i + v_i$;
  para $i \in \{1,\dots,n\}$ **hacer**
    $\overline{c}_{i,j} = c_{i,j} - u_i - v_j$;
    si $\varphi(i) = j$ **entonces**
      $x_{i,j} := 1$,

Algoritmo 5: Detección de vecinos.

Datos: x un nodo del conjunto X,
$A_k \subseteq A$ un conjunto de arcos.
Resultado: Γ_x el conjunto de vecinos de x con respecto a los arcos A_k.
Sea $k := \text{card}(A_k)$ y $\Gamma_x := \varnothing$ **para** $i \in \{1,\dots,k\}$ **hacer**
  si $a_i = (x,y)$, *con* a_i *el* i*-ésimo arco del conjunto* A_k **entonces**
    $\Gamma_x := \Gamma_x \cup \{y\}$,
  si $a_i = (y,x)$, *con* a_i *el* i*-ésimo arco del conjunto* A_k **entonces**
    $\Gamma_x := \Gamma_x \cup \{y\}$.

Algoritmo 6: Detección de componentes conexas.

Datos: n la cardinalidad del conjunto de nodos en la gráfica,
$A_k \subseteq A$ un conjunto de arcos.
Resultado: c la cantidad de componentes conexas en la gráfica (X, A_k).
Sea $N := \{1, \ldots, n\}$, $E := \varnothing$ y $c = 0$;
mientras $N \neq \varnothing$ **hacer**
    Tomamos $x \in N$ y hacemos $E := E \cup \{x\}$;
    mientras $E \neq \varnothing$ **hacer**
        Sea $i \in E$;
        $E := (\Gamma(i) \cap N) \cup (E \setminus \{i\})$;
        $N := N \setminus \{i\}$
    $c := c + 1$.

Algoritmo 7: Índices del conjunto de penalizaciones.

Datos: $\overline{C} \in M_{n\times n}(\mathbb{R})$ una matriz de costos reducidos.
Resultado: p el par ordenado de índices de la variable a ramificar.
Sea $k := 0$, $m := 0$ y $p := (\infty, \infty)$;
para $i \in \{1, \ldots, n\}$ **hacer**
    para $j \in \{1, \ldots, n\}$ **hacer**
        si $\overline{c}_{i,j} = 0$ **entonces**
            $m := \min\{c_{k,j} : k \in \{1, \ldots, n\} \setminus \{i\}\} + \min\{c_{i,l} : l \in \{1, \ldots, n\} \setminus \{j\}\}$;
            si $m < \infty$ y $k < m$ **entonces**
                $k := m$ y $p = (i, j)$

Algoritmo 8: Factibilidad de la solución.

Datos: $X \in M_{n\times n}(\mathbb{R})$ una matriz de asignación.
Resultado: δ un booleano tal que es 1 si y sólo si X es una solución factible.
Sea $\delta = 1$, $a := 0$, $i := 1$ y $j := 1$;
mientras $\delta = 1$ *o* $i \neq 1$ **hacer**
    $\delta = 0$;
    mientras $x_{i,j} = 0$ **hacer**
        $j := j + 1$;
    $i := j$, $j := 1$ y $a := a + 1$;
si $a = n$ **entonces**
    $\delta = 1$.

Algoritmo 9: Ramificación y acotamiento para el PAV.

Datos: $C \in M_{n\times n}(\mathbb{R})$ una matriz de costos.
Resultado: $X \in M_{n\times n}(\mathbb{R})$ una matriz que contiene a un tour óptimo, z el valor óptimo de la instancia asociada a C.
Definimos a $q := 2$, $z_0 := \infty$ y creamos al primer nodo del árbol con la etiqueta #1 con etiqueta **No terminal**;
Sean $w^{(1)}, X^{(1)}, \overline{C}^{(1)}$ el valor óptimo, el óptimo y la matriz de costos reducidos respectivamente obtenidos al aplicar el algoritmo húngaro a C;
Sea $\delta^{(1)}$ el booleano obtenido al aplicar la prueba de factibilidad a X;
mientras *haya un nodo con etiqueta* ***No terminal*** *y sin ramificar* **hacer**
  Sea $p = \ell$ tal que $w^{(\ell)} = \min\{w^{(\alpha)} : \text{el nodo } \#\alpha \text{ es } \textbf{No terminal} \text{ y no se ha ramificado sobre él}\}$;
  si *las variables de ramificación del nodo #p igualadas a 1 forman un ciclo no hamiltoniano* **entonces**
    Etiquetamos al nodo $\#p$ como **No factible**.
  si no, si $\delta^{(p)} = 1$ *y* $w^{(p)} \leq z_0$ **entonces**
    Hacemos $z_0 := w^{(p)}$ y etiquetamos al nodo $\#p$ como **Candidato**;
  si no, si $w^{(p)} \geq z_0$ **entonces**
    Etiquetamos al nodo $\#p$ como **No prometedor**;
  en otro caso
    Sean (i, j) los índices obtenidos del conjunto de penalizaciones;
    Desde $\#p$, ramificamos a los nuevos nodos $\#q$ y $\#(q+1)$;
    Definimos a las matrices $C^{(q)} := \overline{C}^{(p)}$ y $C^{(q+1)} := \overline{C}^{(p)}$;
    Hacemos $c^{(q)}_{i,j} := \infty$, $c^{(q+1)}_{i,j} := 0$, $c^{(q+1)}_{i,l} := \infty$ y $c^{(q+1)}_{k,j} := \infty$, para $l \in \{1,\ldots,n\}\backslash\{j\}$ y $k \in \{1,\ldots,n\}\backslash\{i\}$. Definimos a $z^{(q)}, X^{(q)}, \overline{C}^{(q)}$ el valor óptimo, el óptimo y la matriz de costos reducidos respectivamente obtenidos al aplicar el algoritmo húngaro a $C^{(q)}$.;
    Definimos a $w^{(q)} := z^{(q)} + w^{(p)}$;
    Definimos a $z^{(q+1)}, X^{(q+1)}, \overline{C}^{(q+1)}$ el valor óptimo, el óptimo y la matriz de costos reducidos respectivamente obtenidos al aplicar el algoritmo húngaro a $C^{(q)}$.;
    Definimos a $w^{(q+1)} := z^{(q+1)} + w^{(p)}$;
    $q := q + 2$
Tomamos a $z := z_0$ y definimos a X como la matriz asociada al valor z.

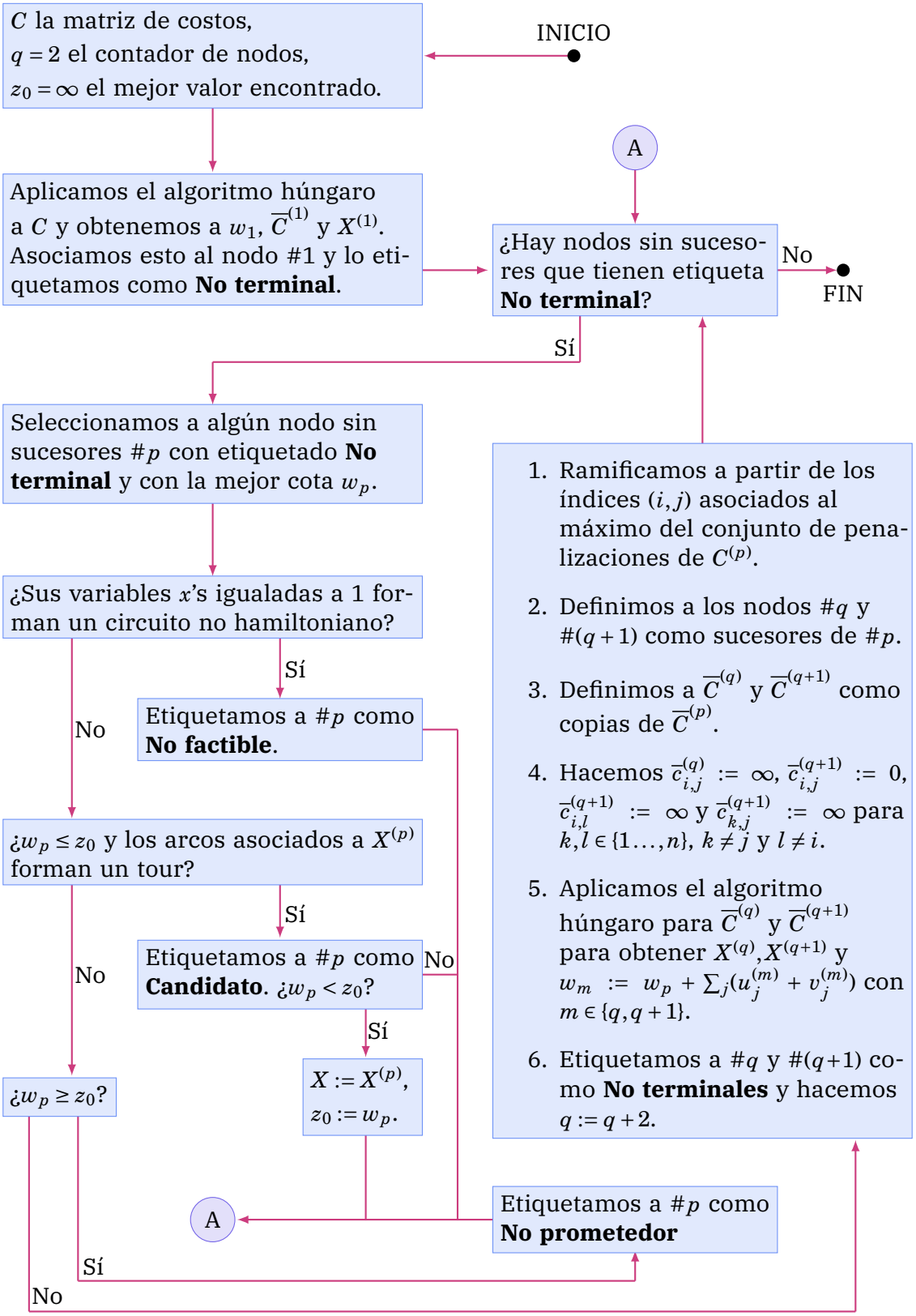


Figura B.1: Diagrama de flujo del algoritmo de ramificación y acotamiento.

B.2. Capítulo 3

Algoritmo 10: Costos con traslapes.

Datos: $\Lambda \subseteq \{\mathtt{A},\mathtt{C},\mathtt{T},\mathtt{G}\}^*$ un conjunto de n lecturas.
Resultado: $C \in M_{n\times n}(\mathbb{R})$ la matriz de costos.
Sea C la matriz tal que $c_{i,j} := \infty$, $\forall i,j \in \{1,\dots,n\}$;
para $i \in \{1,\dots,n\}$ **hacer**
  para $j \in \{1,\dots,n\}\backslash\{i\}$ **hacer**
    $t := 0$, $u := \lambda_i$, $v = \lambda_j$;
    para $k \in \{1,\dots,\text{mín}\{|u|,|v|\}\}$ **hacer**
      si $u_1 \dots u_k = v_{n-k} \dots v_n$ **entonces**
        $t := i$;
      $c_{i,j} := |u| + |v| - 2t$

Algoritmo 11: Distancia de Levenshtein.

Datos: $u, v \in \Gamma^*$ dos cadenas de caracteres
Resultado: $d_{\mathscr{L}}(u,v) \in \mathbb{R}$ la distancia de Levensthein entre u y v.
Sea $L := 0 \in M_{(|u|+1),(|v|+1)}(\mathbb{R})$.;
para $i \in \{1,\dots,|u|+1\}$ **hacer**
  $l_{i,1} := i$;
para $j \in \{1,\dots,|v|+1\}$ **hacer**
  $l_{1,j} := j$;
para $i \in \{2,\dots,|u|+1\}$ **hacer**
  para $j \in \{2,\dots,|v|+1\}$ **hacer**
    $l_{i,j} := \text{mín}\{l_{i,j-1}+1, l_{i-1,j}+1, l_{i-1,j-1}+\mathbb{1}_{u_i,v_j}\}$;
$d_{\mathscr{L}}(u,v) := l_{|u|+1,|v|+1}$;

Algoritmo 12: Costos con traslapes aproximados.

Datos: $\Lambda \subseteq \{\mathtt{A},\mathtt{C},\mathtt{T},\mathtt{G}\}^*$ un conjunto de n lecturas.
Resultado: $C \in M_{n\times n}(\mathbb{R})$ la matriz de costos.
Sea C la matriz tal que $c_{i,j} := \infty$, $\forall i,j \in \{1,\dots,n\}$;
para $i \in \{1,\dots,n\}$ **hacer**
  para $j \in \{1,\dots,n\}\backslash\{i\}$ **hacer**
    $t^* := \infty$, $u := \lambda_i$, $v = \lambda_j$;
    para $k \in \{1,\dots,|u|\}$ **hacer**
      para $l \in \{1,\dots,|v|\}$ **hacer**
        $s := u_{n-k}\dot{s}u_n$, $p := v_1 \dots v_l$;
        $t := |s| + |p| - (1 + \text{mín}\{|s|,|p|\}/\text{máx}\{|s|,|p|\})d_{\mathscr{L}}(s,p)$;
        si $t > t*$ **entonces**
          $t^* = t$;
    $c_{i,j} := |u| + |v| - t^*$;

B.3. Capítulo 4

Algoritmo 13: Evaluación de aptitudes.

Datos: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ una tupla con m ordenamientos de orden n,
$C \in M_{n\times n}(\mathbb{R})$ la matriz de costos.
Resultado: $v \in \mathbb{R}^m$ un vector de aptitudes de la población
P la población ordenada de menor a mayor costo.
Sea $v := 0 \in \mathbb{R}^m$;
para $i \in \{1,\dots,m\}$ **hacer**
    Sea $\sigma := P_i$;
    $v_i := 1/\left(\sum_{j=1}^{n-1} c_{\sigma(j),\sigma(j+1)}\right)$;
Ordenamos a v;
Reordenamos a P con respecto a v;

Algoritmo 14: Índices de selección.

Datos: $v \in \mathbb{R}^n$ el vector de aptitudes de la población,
ε el parámetro de elitismo.
Resultado: $I \in \mathbb{R}^{m-\varepsilon}$ los índices de pobladores a cruzar.
Sea $F := 0 \in \mathbb{R}^m$ y $S := \sum_{i=1}^{n} v_i$;
Hacemos $F_1 := v_1/S$;
para $i \in \{1,\dots,m-1\}$ **hacer**
    $F_{i+1} := F_i + v_{i+1}/S$;
Generamos al conjunto $\{\xi_i\}_{i=1}^{m-\varepsilon}$ de números aleatorios;
Definimos a $I := 0 \in \mathbb{R}^{m-\varepsilon}$;
para $i \in \{1,\dots,m-\varepsilon\}$ **hacer**
    $\eta := 0$;
    mientras $\xi_i < F_\eta$ **hacer**
        $\eta := \eta + 1$;
    $I_i := \eta - 1$

Algoritmo 15: Reproducción MPX.

Datos: $(\sigma_1, \sigma_2) \in \mathscr{S}_n \times \mathscr{S}_n$ dos ordenamientos de orden n.
Resultado: $\varsigma \in \mathscr{S}_n$ un ordenamiento de orden n.
Sean $\xi_1, \xi_2 \sim \text{UnifDisc}(\{1, \ldots, n\})$ v.a.i.i.d. y $\varsigma := 0 \in \mathbb{R}^n$;
Definimos a $\mu := \text{mín}\{\xi_1, \xi_2\}$, $M := \text{máx}\{\xi_1, \xi_2\}$ y $V := \varnothing$;
para $i \in \{\mu, \ldots, M\}$ **hacer**
    $\varsigma(i) := \sigma_1(i)$;
    $V := V \cup \{\varsigma(i)\}$;
para $i \in \{1, \ldots, n\}$ **hacer**
    para $j \in \{1, \ldots, n\} \setminus \{\mu, \ldots, M\}$ **hacer**
        si $\sigma_2(i) \notin V$ **entonces**
            $\varsigma(j) := \sigma_2(i)$;
            $V := V \cup \{\varsigma(j)\}$;

Algoritmo 16: Reproducción de la población.

Datos: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ una tupla con m ordenamientos de orden n,
ε el parámetro de elitismo,
$I \in \mathbb{R}^{m-\varepsilon}$ los índices de pobladores a cruzar.
Resultado: P' un conjunto de m ordenamientos de orden n.
para $i \in \{1, \ldots, \epsilon\}$ **hacer**
    $P'_i := P_i$;
para $i \in \{1, \ldots, m - \varepsilon - 1\}$ **hacer**
    $P'_{i+\varepsilon}$ es el descendiente del par $(P_{I(i)}, P_{I(i+1)})$;
P'_m es el descendiente de reproducir al par $(P_{m-\varepsilon}, P_1)$;

Algoritmo 17: Mutación.

Datos: $\sigma \in \mathscr{S}_n$ un ordenamiento de orden n.
Resultado: $\sigma \in \mathscr{S}_n$ un ordenamiento de orden n.
Sean $\xi_1, \xi_2 \sim \text{UnifDisc}(\{1, \ldots, m\})$ v.a.i.i.d.;
Hacemos $h := \sigma(\xi_1)$, $\sigma(\xi_1) := \sigma(\xi_2)$ y $\sigma(\xi_2) := h$.

Algoritmo 18: Mutación de la población.

Datos: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ un vector con m ordenamientos de orden n,
p una probabilidad de mutación.
Resultado: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ un vector con m ordenamientos de orden n.
Generamos al conjunto $\{\xi_i\}_{i=1}^{m}$ de números aleatorios.;
para $i \in \{1, \ldots, m\}$ **hacer**
    si $\xi_i \leq p$ **entonces**
        P_i es el resultado de mutar a P_i

Algoritmo 19: Producción de nueva generación.

Datos: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ un tupla con m ordenamientos de orden n,
$C \in M_{n\times n}(\mathbb{R})$ la matriz de costos,
ε el parámetro de elitismo,
$p \in [0,1]$ la probabilidad de mutación.
Resultado: $P \in \prod_{i=1}^{m} \mathscr{S}_n$ una vector con m ordenamientos de orden n.
Obtenemos el vector de aptitudes v y la población ordenada con respecto a él;
Sometemos a la población P al proceso de selección para obtener los índices I;
Reproducimos a la población P considerando al conjunto I y al parámetro ε;
Mutamos a la población P con probabilidad p;

Algoritmo 20: Genético.

Datos: $m \in \mathbb{Z}^+$ el tamaño de la población,
$C \in M_{n\times n}(\mathbb{R})$ la matriz de costos,
$\varepsilon \in \{0,\dots,m\}$ el parámetro de elitismo,
$p \in [0,1]$ la probabilidad de mutación,
$g \in \mathbb{Z}^+$ el número de generaciones a simular.
Resultado: $\sigma \in \mathscr{S}_n$ el mejor ordenamiento de orden n con respecto a C,
z el mejor valor encontrado de la instancia.
Definimos $\sigma \in \mathscr{S}_n$ un ordenamiento arbitrario y $z := \infty$. Generamos a $P \in \prod_{i=1}^{m} \mathscr{S}_n$ una tupla con m ordenamientos de orden n;
para $i \in \{1,\dots,g\}$ **hacer**
    Modificamos a P como una nueva generación con los parámetros m, C, ε y p.;
    Obtenemos el vector de aptitudes v y la población ordenada con respecto a él;
    si $v_1 < z$ **entonces**
        $z := v1$;
        $\sigma := P_1$;

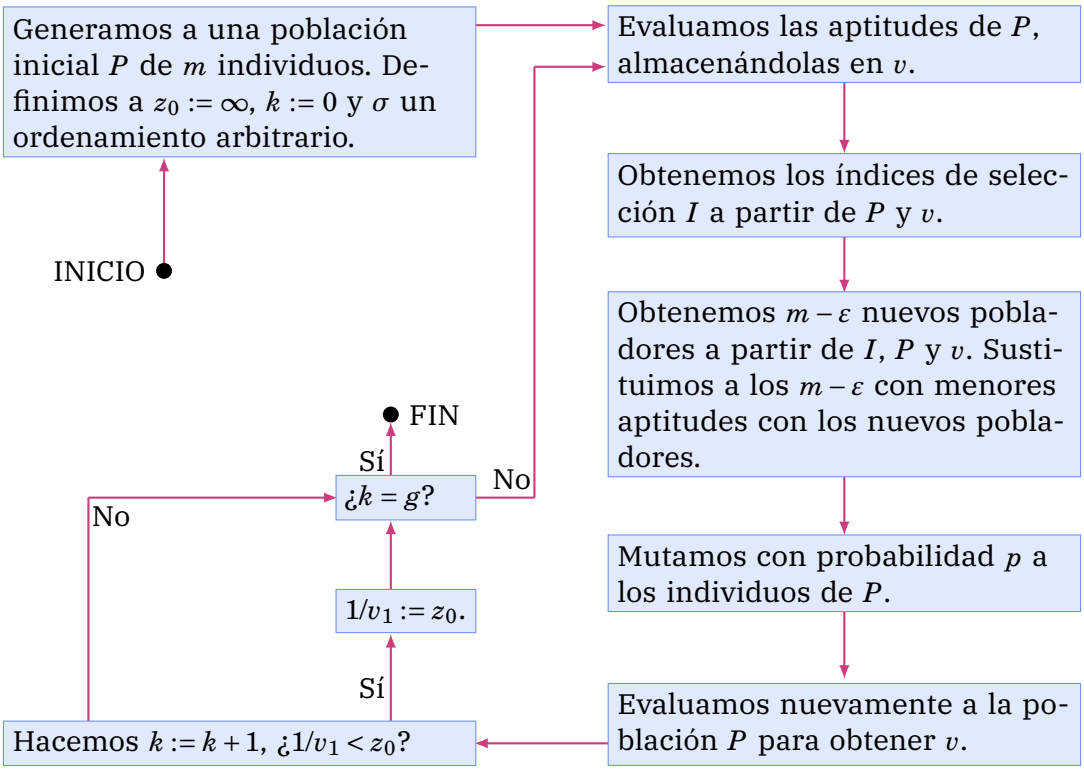


Figura B.2: Diagrama de flujo del algoritmo genético.

Referencias y bibliografía

[1] D. Gusfield, *Integer Linear Programming in Computational and Systems Biology*. Cambridge University Press, 2019.

[2] H. Curtis, H. S. Barnes, A. Schnek y A. Massarini, *Biología*, 7.ª ed., E. M. Panamericana, ed. 2008, págs. 172-204.

[3] J. M. Heather y B. Chain, «The sequence of sequencers: The history of sequencing DNA», *Genomics*, vol. 107, n.º 1, págs. 1-8, 2016. doi: 10.1016/j.ygeno.2015.11.003.

[4] B. Langmead. (2015). Sequencers give pieces to genomic puzzles, Youtube, dirección: https://youtu.be/X307VyAeHfI.

[5] C. J. Avers, *Biología celular*, 2.ª ed., G. E. Iberoamérica, ed., trad. por I. de León y A. Pérez. 1991, págs. 27-113.

[6] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts y P. Walter., *Molecular Biology of the Cell*, 4.ª ed. Garland Science, 2002, págs. 1275-1291.

[7] D. L. Applegate, R. E. Bixby, V. Chvatál y W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[8] *Aeroméxico*, aeromexico.com/es-mx/, Consultado en 2021-01-21, los tiempos fueron ligeramente modificados.

[9] G. Dantzig, D. Fulkerson y S. Johnson, «Solution of a Large-Scale Traveling-Salesman Problem», *Journal of the Operations Research Society of America*, vol. 2, n.º 4, págs. 393-410, 1954. doi: 10.1287/opre.2.4.393.

[10] C. E. Miller, A. W. Tucker y R. A. Zemlin, «Integer Programming Formulation of Traveling Salesman Problems», *Journal of the ACM*, vol. 7, n.º 4, págs. 326-329, 1960. doi: 10.1145/321043.321046.

[11] R. Bukard, M. Dell'Amico y S. Martello, *Assignment Problems*, 1.ª ed. SIAM, 2009.

[12] M. Conforti, G. Cornuejols y G. Zambelli, *Integer Programming*, 1.ª ed., S. I. Publishing, ed. 2014, págs. 99, 100, 131 y 132.

[13] M. C. Hernández–Ayuso, *Introducción a la Programación Lineal*. Prensas de Ciencias, 2010.

[14] N. Veldt. (2016). Totally Unimodular Matrices in Linear Programming, Youtube, dirección: https://youtu.be/Fmjy74c-R-I.

[15] G. Allaire y S. M. Kaber, *Numerical Lineal Algebra*, 1.ª ed., Springer, ed. 2008.

[16] J. D. C. Little, K. G. Murty, D. W. Sweeney y C. Karel, «An Algorithm for the Traveling Salesman Problem», *Operations Research*, vol. 11, n.º 6, págs. 972-989, 1963. doi: 10.1287/opre.11.6.972.

[17] M. C. Hernández–Ayuso, *Introducción a la Teoría de Redes*. Sociedad Matemática Mexicana, 2005.

[18] F. Pfeiffer, C. Gröber, M. Blank, K. Händler, M. Beyer, J. L. Schultze y G. Mayer, «Systematic evaluation of error rates and causes in short samples in next-generation sequencing», *Artificial Intelligence Review*, vol. 8, págs. 1-14, 2018. doi: 10.1038/s41598-018-29325-6.

[19] T. Massingham y N. Goldman, «All your base: a fast and accurate probabilistic approach to base calling», *Genome Biology*, vol. 13 R13, feb. de 2012. doi: 10.1186/gb-2012-13-2-r13.

[20] J. Peirce, *Illumina: MiSeq: Imaging and Base Calling*, support.illumina.com/content/dam/illumina-support/courses/MiSeq_Imaging_and_Base_Calling/story_html5.html, Consultado en 2021-03-20.

[21] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer y P. M. Rice, «The Sanger FASTQ format for sequences with quality scores, and the Solexa/Illumina FASTQ variants», *Nucleic acids research*, vol. 38, págs. 1767-1771, dic. de 2009. doi: 10.1093/nar/gkp1137.

[22] M. Sipser, *Introduction to the Theory of Computation*, 3.ª ed. Thomson South-Western, 2012, págs. 273-322.

[23] A. Lapidus, «Genome Sequence Databases: Sequencing and Assembly», en. dic. de 2015, págs. 196-210, isbn: 9780128012383. doi: 10.1016/B978-0-12-801238-3.02495-8.

[24] C. Mackenzie, *Coded Character Sets, History and Development*. Addison-Wesley Publishing Company, Inc., 1980.

[25] *El codigo ASCII*, elcodigoascii.com.ar, Consultado en 2021-03-16.

[26] K.-J. Räihä y E. Ukkonen, «The shortest common supersequence problem over binary alphabet is NP-complete», *Theoretical Computer Science*, vol. 16, n.º 2, págs. 187-198, 1981. doi: 10.1016/0304-3975(81)90075-x.

[27] R. S. Boyer y J. S. Moore, «A fast string searching algorithm», *Communications of the ACM*, vol. 20, n.º 10, págs. 762-772, 1977. doi: 10.1145/359842.359859.

[28] J. Simpson. (2017). Fundamentals of Genome Assembly, Youtube, dirección: https://www.youtube.com/watch?v=5wvGapmA5zM.

[29] *Babraham Institute: FastQC*, https://www.bioinformatics.babraham.ac.uk/projects/fastqc/, Consultado en 2021-04-10.

[30] J. Schröder, H. Schröder, S. J. Pgulisi y R. Sinha, «SHREC: a short-read error correction method», *Bioinformatics*, vol. 25, n.º 17, págs. 2157-2163, 2009. doi: 10.1093/bioinformatics/btp279.

[31] Y. Liu, B. Schmidt y D. L. Maskell, «DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI», *BMC Bioinformatics*, vol. 12, n.º 85, págs. 1-13, 2011. doi: 10.1186/1471-2105-12-85.

[32] G. Navarro, «A Guided Tour to Approximate String Matching», vol. 33, n.º 1, págs. 31-88, 2001. doi: 10.1145/375360.375365.

[33] J. Erickson, *Algorithms*. University of Illinois, 2013, cap. 5, págs. 7-11.

[34] R. Karp, *Complexity of Computer Computations*, R. Miller y J. Thatcher, eds. Plenum Press, 1972, págs. 85-103.

[35] *OpenDSA Data Structures and Algorithms Modules Collection, Reduction of 3-SAT to Hamiltonian Cycle*, https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/threeSAT_to_hamiltonianCycle.html, Consultado en 2020-12-13.

[36] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela y M. Protasi, *Complexity and Approximation*, 1.ª ed., S.-V. B. Heidelberg, ed. 1999, págs. 8-33.

[37] *Clay Mathematics Institute: P vs NP Problem*, www.claymath.org/millennium-problems/p-vs-np-problem, Consultado en 2020-12-26.

[38] G. Polya, *How to solve it*, 2.ª ed., Princeton, ed. 2014.

[39] D. F. Mansfield y N. Wildberger, «Plimpton 322 is Babylonian exact sexagesimal trigonometry», *Historia Mathematica*, vol. 44, n.º 4, págs. 395-419, 2017. doi: 10.1016/j.hm.2017.08.001.

[40] F.-S. Wang y L.-H. Chen, «Heuristic Optimization», en *Encyclopedia of Systems Biology*, W. Dubitzky, O. Wolkenhauer, K.-H. Cho y H. Yokota, eds. New York, NY: Springer New York, 2013, págs. 885-885. doi: 10.1007/978-1-4419-9863-7_411.

[41] B. Melian, J. Moreno-Pérez y J. Moreno-Vega, «Metaheuristicas: Una visión global», *Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial, ISSN 1137-3601, null 7, Nº. 19, 2003, pags. 7-28*, vol. 7, ene. de 2003.

[42] E. Tardos y J. Kleinberg, *Algorithm Design*, 1.ª ed. PEARSON Addison Wesley, 2005, págs. 707-708.

[43] A. P. Alves da Silva y D. M. Falcão, «Fundamentals of Genetic Algorithms», en *Modern Heuristic Optimization Techniques*. John Wiley & Sons, Ltd, 2008, cap. 2, págs. 25-42.

[44] P. Larrañaga, C. Kuijpers, I. I. R. Murga y S. Dizdarevic, «Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators», *Scientific Reports*, vol. 13, págs. 129-170, 1999. doi: 10.1023/A:1006529012972.

[45] G. Pagès, *Numerical Probability*, 1.ª ed. Springer, 2018, págs. 5-8.

[46] G. van Rossum y F. L. Drake, *Python 3 Reference Manual*. CreateSpace, 2009.

[47] S. Ekins, J. Mestres y B. Testa, «In silico pharmacology for drug discovery: methods for virtual ligand screening and profiling», *British Journal of Pharmacology*, vol. 152, n.º 1, págs. 9-20, 2007. doi: 10.1038/sj.bjp.0707305.

[48] S. Alosaimi, A. Bandiang, N. van Biljon, D. Awany, P. K. Thami, M. S. Tchamga, A. Kiran, O. Messaoud, R. I. M. Hassan, J. Mugo, A. Ahmed, C. D. Bope, I. Allali, G. K. Mazandu, N. J. Mulder y E. R. Chimusa, «A broad survey of DNA sequence data simulation tools», *Briefings in Functional Genomics*, págs. 1-11, 2019. doi: 10.1093/bfgp/elz033.

[49] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke y T. E. Oliphant, «Array programming with NumPy», *Nature*, vol. 585, n.º 7825, págs. 357-362, 2020. doi: 10.1038/s41586-020-2649-2.

[50] *Escherichia phage Lambda, complete genome*, www.ncbi.nlm.nih.gov/nuccore/J02459.1?report=fasta, Consultado en 2020-12-23.

[51] *Illumina: Specifications for the MiSeq System*, www.illumina.com/systems/sequencing-platforms/miseq/specifications.html, Consultado en 2021-04-04.

[52] T. Kato y M. Hasegawa, «Performance of heuristic methods driven by chaotic dynamics for ATSP and applications of DNA fragment assembly», *Bioinformatics*, vol. 25, n.º 17, págs. 2157-2163, 2009. doi: 10.1093/bioinformatics/btp279.

[53] B. Ma, «Why greed works for shortest common superstring problem», *Theoretical Computer Science*, vol. 410, n.º 51, 2009. doi: 10.1016/j.tcs.2009.09.014.

[54] P. A. Pevzner, H. Tang y M. S. Waterman, «An Eulerian path approach to DNA fragment assembly», *IEICE*, vol. 2, n.º 4, págs. 485-496, 2011. doi: 10.1587/notla.2.485.

[55] H. Fleischner, *Eulerian Graphs and Related Topics*, 1.ª ed. North–Holland, 1991, vol. 1, pág. VI.33.

[56] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov y E. S. Lander, «ARACHNE: A Whole-Genome Shotgun Assembler», *Genome Research*, vol. 12, n.º 1, págs. 177-189, 2002. doi: 10.1101/gr.208902.

[57] N. B. Nsira, T. Lecroq y M. Elloumi, «A fast Boyer-Moore type pattern matching algorithm for highly similar sequences», *International Journal of Data Mining and Bioinformatics*, vol. 13, n.º 3, 2015. doi: 10.1504/IJDMB.2015.072101.

[58] B. Weinhold, «Epigenetics: The Science of Change», *Environ Health Perspect*, vol. 114, n.º 3, A160-A167, 2006. doi: 10.1289/ehp.114-a160.

[59] H. Karloff, *Linear Programming*, 1.ª ed. Birkhäuser Basel, 1991.

[60] I. R. Shafarevich y A. Remizov, *Linear Algebra and Geometry*, 1.ª ed. Springer, 2012, pág. 56.