UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA, MANAGUA
UNAN - MANAGUA

CENTRO UNIVERSITARIO REGIONAL MATAGALPA

CUR - MATAGALPA

TEMA:

DISEÑO DE ALGORITMOS HEURISTICOS Y METAHEURISTICOS EFICIENTES PARA EL PROBLEMA SBPR (SORTING PERMUTATIONS BY PREFIX REVERSALS)

TÉSIS PARA OPTAR AL GRADO DE DOCTOR EN MATEMÁTICA APLICADA

AUTOR:

MSC. WILMER JOSÉ PALACIOS LOPEZ

TUTOR:

DR. FERNANDO JOSÉ HERNÁNDEZ GÓMEZ

MATAGALPA, MAYO, 2024

¡UNIVERSIDAD DEL PUEBLO Y PARA EL PUEBLO!

La mayoría de las personas, Paulino, se queja unánime de la malicia de la naturaleza que nos ha engendrado para un período de vida muy escaso, porque el espacio de tiempo que se nos otorga transcurre tan veloz, tan rápidamente que, con excepción de unos pocos, casi todos los demás quedan incapacitados cuando apenas se están preparando para vivir. Y ante este mal, que según creen es general, no se lamentan solamente la masa y el vulgo necio, también este mismo sentimiento ha hecho que personajes eminentes se quejen.

No tenemos un tiempo escaso, Paulino, sino que desperdiciamos mucho. La vida es bastante larga para la realización de las empresas más importantes y se nos ha otorgado con generosidad, si se emplea bien toda ella. Pero si se derrocha en la ostentación y la dejadez, si no se gasta en nada bueno, cuando al fin nos acosa el inevitable momento final, nos damos cuenta de que se nos ha pasado una vida que no supimos que estaba pasando.

La cosa es así: no recibimos una vida corta sino que la hacemos corta; no somos mendigos de ella sino derrochadores. Tal como unas riquezas cuantiosas y principescas que cuando caen en manos de un mal amo, en instante se pierden, y al revés, cuando, pese a ser escasas, se entregan a un buen custodio, crecen al emplearlas, igualmente la existencia se le expande mucho más a quien bien la organiza.

Sobre la brevedad de la vida
Séneca

El arte de programar es el arte de organizar la complejidad.

Notes on Structured Programming
E.W. Dijkstra

AGRADECIMIENTOS

En primer lugar, le doy gracias a Dios Padre Rey del Universo, por darme la sabiduría, paciencia y amor, pero sobre todo por haber puesto en mi camino a aquellas personas que son mi soporte y mi compañía en esta nueva etapa de mi vida.

A mis padres: Félix Palacios y Josefa López, por su amor Universal. Hoy estoy aquí por ellos.

A la que considero mi segunda familia: Dra. Guisselle Martínez y a su hijo Diego Fernando, cuyo apoyo en estos últimos años ha sido invaluable. Acuño su amistad como mi más grande tesoro.

A mi tutor de tesis: Dr. Fernando Hernández por su apoyo constante, consejos y paciencia brindada durante mi formación académica. Gracias por brindarme su tiempo y observaciones de forma crítica. Todo esto ha sido vital para la elaboración del informe final.

A todos mis profesores de la carrera de Ingeniería en CUR Matagalpa y del Programa en Matemática Aplicada, por sus sabios consejos y enseñanzas de vida compartidos a lo largo de todos estos años.

Agradezco plenamente a todas y cada una de las personas que han vivido estas mismas experiencias conmigo y que han sido mis cómplices durante la realización de esta tesis, con todos los altos y bajos que el proceso conlleva. No necesito nombrarlos. Descubrí inesperadamente que estas son batallas que se luchan en conjunto. Creo que las ganamos casi todas. Muchísimas gracias.

TRAYECTORIA DEL INVESTIGADOR

Wilmer José Palacios López, obtuvo su ingeniería en Ciencias de la Computación por la Universidad Nacional Autónoma de Nicaragua (UNAN-Managua, 2019). Es Máster en Matemática Aplicada en el Centro Universitario Regional Matagalpa, de la Universidad Nacional Autónoma de Nicaragua (CUR-Matagalpa, 2021). Actualmente, se encuentra finalizando el programa de doctorado en Matemática Aplicada, coordinado por la Universidad Nacional Autónoma de Nicaragua, CUR Matagalpa. Desarrollando Tesis Doctoral, en Optimización Combinatoria, bajo la tutoría del Dr. Fernando José Hernández Gómez, específicamente línea de investigación sobre el problema de Ordenamiento de Permutaciones, en los que se diseñaron e implementaron algoritmos heurísticos y metaheurísticos eficientes para resolver el MIN-SBPR.

Además, su motivación y compromiso con la investigación se ha reflejado en sus contribuciones académicas, como la publicación de trabajos significativos "Incidencia de los estudios sobre ordenamiento genómico en la secuenciación del genoma humano" (2023) y "Una revisión y clasificación de los algoritmos para el problema SBPR (Sorting Permutations By Prefix Reversals)" (2023). De la misma forma, ha sido invitado como conferencista en eventos académicos de prestigio, donde ha presentado sus investigaciones, como "Diseño de algoritmos heurísticos y metaheurísticos eficientes para el problema SBPR (Sorting Permutations by Prefix Reversals) - Keiser University" (2023) and "Metaheuristics for Sorting Permutations by Prefix Reversals (SPBR) - XXIII International Symposium of Mathematical Methods Applied to Sciences" (2021).

Por otro lado, también ha trabajado como profesor horario en la Universidad Juan Pablo II de Matagalpa (ahora Universidad Padre Gaspar García Laviana) dictando cursos sobre Teoría de Compiladores, Simulación de sistemas y Programación algorítmica con Python. A su vez ha sido docente horario de Cálculo y Estadística en la Universidad Nacional Multidisciplinaria Ricardo Morales Avilés en Matagalpa así como tutor de tesis de pregrado para las carreras de Ingeniería Industrial e Ingeniería en Sistemas de Información.

Actualmente se destaca como analista de datos y como docente horario en la Universidad Nacional Autónoma de Nicaragua UNAN−Managua, impartiendo cursos relacionados con bases de datos y desarrollo de software en general.

CARTA AVAL

El **Dr. Fernando José Hernández Gómez**, es docente externo de la Facultad de Ciencias e Ingeniería Departamento de Matemática y Estadística de la Universidad Nacional Autónoma de Nicaragua y Docente Titular Departamento de Matemática de Keiser University Latin American Campus.

CERTIFICA, que la presente memoria de investigación:

Diseño de Algoritmos Heurísticos y Metaheurísticos eficientes para el problema SBPR *(Sorting Permutations By Prefix Reversals)*

Ha sido realizada bajo su dirección en el PROGRAMA DE DOCTORADO EN MATEMÁTICA APLICADA por el **Máster Wilmer José Palacios López**, y constituye su tesis para optar al grado de Doctor en Matemática Aplicada, representando su contribución sustancial para la obtención del grado de Doctor. Es un estudiante altamente calificado y motivado, cuyas habilidades analíticas y destrezas en Matemática Aplicada y Ciencias de la Computación han sido fundamentales para diseñar e implementar algoritmos heurísticos y metaheurísticos existentes e innovadores. Estos enfoques, diseñados para mejorar las soluciones existentes en la literatura especializada, representan un avance significativo en el campo de estudio en Nicaragua.

Y para que así conste, en cumplimiento con la normativa vigente de posgrado, autoriza su presentación ante el Centro Universitario Regional (UNAN-CUR-Matagalpa) para que pueda ser tramitada su lectura y defensa pública.

Managua, Nicaragua, 27 de Marzo de 2024.

El DIRECTOR DE LA TESIS

27/3/24

Dr. Fernando José Hernández Gómez

ABREVIATURAS

AA : *(en inglés)* Approximation Algorithm.

ADN : Ácido Desoxirribonucleico.

AI : *(en inglés)* Artificial Intelligence.

GA : *(en inglés)* Genetic Algorithm.

HA : *(en inglés)* Heuristic Algorithm.

iGA : *(en inglés)* Improved Genetic Algorithm.

JCR : *(en inglés)* Journal Citation Report.

LB : *(en inglés)* Lower Bound.

MHA : *(en inglés)* Metaheuristic Algorithm.

ML : *(en inglés)* Machine Learning.

NP : *(en inglés)* Non deterministic polynomial time.

NP-Hard : *(en inglés)* Non Deterministic Polynomial Time hardness.

SBPR : *(en inglés)* Sorting Permutations by Prefix Reversals.

SBR : *(en inglés)* Sorting Permutations by Reversals.

SI : *(en inglés)* Swarm intelligence.

ABSTRACT

Sorting permutations by prefix reversals (SBPR) is a classical combinatorial optimization problem whose main objective is to sort a permutation of n elements reversing the leftmost (prefix) blocks of that permutation. Technically, a given permutation π must be transformed into an identity permutation named id. Similarly, it could be compared in magnitude to a stack of pancakes that must be rearranged with a spatula by inserting it at any point and inverting its order so that after several iterations the stack is ordered.However, since 2012 it has been shown that this problem belongs to the NP-Hard class, which makes it unfeasible in practical terms to construct an optimal solution in polynomial time or less. In fact, in recent years, the academic community has tried to use heuristic and metaheuristic approximation algorithms that provide better results.In this research, heuristic and metaheuristic algorithms were designed and implemented, based on the criteria proposed by the reviewed literature. Also, a family of life-inspired metaheuristics was proposed that have proven to be very efficient in finding solutions in large search spaces. The algorithms used during the implementation of this research work were written in Python, using the Google Colab tool. In addition, the basic concepts related to modern permutation theory and its elementary ordering methods were reviewed. The evaluation of the algorithms was performed using the set of results provided by the literature of the scientific community in the last 10 years. We showed the advantages and disadvantages of using the different methods proposed and ultimately which of them will prove to be the most appropriate depending on the context in which the problem is placed. Finally, this work contributes greatly to the dissemination, extension, and improvement of the study of combinatorial optimization from the point of view of the implementation and evaluation of algorithms.

Keywords: *Sorting permutations, Heuristics algorithms, Metaheuristics algorithms, Combinatorial optimization*

RESUMEN

El ordenamiento de permutaciones por reversión de prefijos (SBPR) es un problema clásico de optimización combinatoria cuyo objetivo principal es ordenar una permutación de n elementos invirtiendo los bloques más a la izquierda (prefijos) de dicha permutación. Técnicamente, una permutación dada π debe transformarse en una permutación identidad llamada ι. De forma similar, podría compararse en magnitud a una pila de tortitas que hay que reordenar con una espátula introduciéndola en cualquier punto e invirtiendo su orden para que tras varias iteraciones la pila quede ordenada. Sin embargo, desde 2012 se ha demostrado que este problema pertenece a la clase NP-Hard, lo que hace inviable en términos prácticos la construcción de una solución óptima en tiempo polinómico o inferior. De hecho, en los últimos años, la comunidad académica ha tratado de utilizar algoritmos heurísticos y metaheurísticos de aproximación que proporcionen mejores resultados. En esta investigación se diseñaron e implementaron algoritmos heurísticos y metaheurísticos, basados en los criterios propuestos por la literatura revisada. Además, se propuso una familia de metaheurísticas inspiradas en la vida que han demostrado ser muy eficientes en la búsqueda de soluciones en grandes espacios de búsqueda. Los algoritmos utilizados durante la implementación de este trabajo de investigación fueron escritos con Python, utilizando la herramienta Google Colab. Además, se revisaron los conceptos básicos relacionados con la teoría moderna de permutaciones y sus métodos elementales de ordenación. La evaluación de los algoritmos se realizó utilizando el conjunto de resultados proporcionados por la literatura de la comunidad científica en los últimos 10 años. Se mostraron las ventajas y desventajas del uso de los diferentes métodos propuestos y, en última instancia, cuál de ellos resultará ser el más adecuado en función del contexto en el que se sitúe el problema.

Palabras clave: *Algoritmos heurísticos, Algoritmos metaheurísticos, Optimización combinatoria, Ordenamiento de permutaciones*

UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA, MANAGUA
UNAN - MANAGUA

CENTRO UNIVERSITARIO REGIONAL MATAGALPA

CUR - MATAGALPA

TEMA:

DISEÑO DE ALGORITMOS HEURISTICOS Y METAHEURISTICOS EFICIENTES PARA EL PROBLEMA SBPR (SORTING PERMUTATIONS BY PREFIX REVERSALS)

TÉSIS PARA OPTAR AL GRADO DE DOCTOR EN MATEMÁTICA APLICADA

AUTOR:

MSC. WILMER JOSÉ PALACIOS LOPEZ

TUTOR:

DR. FERNANDO JOSÉ HERNÁNDEZ GÓMEZ

MATAGALPA, MAYO, 2024

¡UNIVERSIDAD DEL PUEBLO Y PARA EL PUEBLO!

ÍNDICE

AGRADECIMIENTOS I

TRAYECTORIA DEL INVESTIGADOR II

CARTA AVAL III

ABREVIATURAS IV

ABSTRACT V

RESUMEN VI

1. INTRODUCCIÓN **2**

1.1. Planteamiento del problema 2

1.2. Antecedentes 4

1.3. Justificación 10

1.4. Área del conocimiento 12

1.5. Tema y objetivos de investigación 13

1.5.1. Tema de investigación 13

1.5.2. Objetivo General 13

1.5.3. Objetivos Específicos 13

1.6. Contribuciones 13

1.7. Publicaciones y conferencias asistidas 15

1.8. Organización de la tesis 15

2. RESEÑA HISTÓRICA DEL PROBLEMA SBPR **19**

2.1. Algoritmos metaheurísticos 19

2.2. Algoritmos heurísticos 21

3. FUNDAMENTOS MATEMÁTICOS DEL PROBLEMA SBPR **25**

3.1. Permutaciones 25

3.1.1. Operaciones 27

3.1.2. Formulaciones 27

3.1.3. Problema SBPR 28

3.1.4. Puntos de interrupción 29

3.1.5. Trozos 29
3.1.6. Grafos con puntos de interrupción 30
3.2. Algoritmos heurísticos 31
3.2.1. Algoritmo de ventana corrediza 31
3.2.2. Algoritmo de búsqueda anticipada 32
3.3. Algoritmos metaheurísticos 33
3.3.1. Algoritmo genético 36
3.3.2. Algoritmo memético 41
3.3.3. Algoritmo de colonia de hormigas 42
3.4. Ordenes de complejidad computacional 48
3.4.1. Análisis de casos 48
3.4.2. Notación asintótica 50
3.5. MIN-SBPR es NP-Hard 51
3.5.1. Conceptos y notación 53

4. ALGORITMOS HEURÍSTICOS Y METAHEURÍSTICOS DEL PROBLEMA SBPR 57
4.1. Heurísticas 57
4.1.1. Algoritmo de dias2014 57
4.1.2. Algoritmo de brito2019 60
4.2. Metaheurísticas 63
4.2.1. Algoritmo de soncco2013 63
4.2.2. Algoritmo de al2014 66
4.2.3. Algoritmo de soncco2014 69

5. METODOLOGÍA APLICADA 72
5.1. Metodología aplicada 72
5.2. Modalidad de la investigación 74
5.3. Herramientas computacionales 76

6. IMPLEMENTACIÓN DE ALGORITMOS Y RESULTADOS COMPUTACIONALES 79
6.1. Resultados de la literatura 79
6.2. Resultados computacionales obtenidos 84
6.2.1. Algoritmos heurísticos 84

6.2.2. Algoritmos metaheurísticos 87
6.3. Algoritmos propuestos 96
6.3.1. Algoritmo heurístico 96
6.3.2. Algoritmos metaheurísticos 99

7. CONCLUSIONES **107**

REFERENCIAS **110**

ANEXOS **117**

Índice de tablas

1. Algoritmos a implementar en esta tesis 23
2. Resultados obtenidos al generar la base de datos con el script propuesto para el algoritmo dias2014 86
3. Resultados de algoritmos metaheurísticos 105
4. Resultados de algoritmos heurísticos 105
5. Algoritmos con mejores resultados en complejidad computacional . . 105

Índice de figuras

1. *Familia de problemas sobre ordenamiento de permutaciones* 3
2. *Bill Gates, fundador de Microsoft* 6
3. *Resumen del mapeo completo del genoma humano* 17
4. *Dr. Laurent Bulteau en 2014* 20
5. *Diagrama de flujo de ventana corrediza* 22
6. *Grafo de puntos de interrupción* 30
7. *Funcionamiento del algoritmo de ventana corrediza* 32
8. *Clasificación de los algoritmos metaheurísticos* 35
9. *Diagrama de flujo básico de un GA* 37
10. *GA - Paso 1* 38
11. *GA - Paso 2* 38
12. *GA - Paso 3* 39
13. *GA - Paso 4* 39
14. *GA - Paso 5* 40
15. *GA - Paso 6* 40
16. *Resultados del GA* 41
17. *Elementos básico de un algoritmo memético* 42
18. *Comparativa entre GA y MA* 43
19. *Disciplinas relacionadas con ACO* 44
20. Un ejemplo con hormigas reales 45
21. *Marco de trabajo de ACO* 46
22. *Funciones más comunes en el análisis de algoritmos* 49
23. Orden de complejidad: Omega grande (Ω), theta grande (Θ) y o grande (O). 52
24. *Ejemplos del crecimiento asintótico de funciones comunes* 52
25. *Compilación de todas las propiedades gadget* 55
26. *Metodología de trabajo en esta investigación* 73
27. *Conexión entre la secuenciación del genoma humano y el origen del problema SBPR* 75
28. *Funcionamiento de Google Colab* 76
29. Especificaciones de hardware en donde se ejecutaron algunos de los algoritmos de esta tesis 77
30. *Algoritmo de ventana corrediza* 79

31. *Algoritmo de búsqueda anticipada* 80
32. *Algoritmo genético* 81
33. *Algoritmo de colonia de hormigas* 82
34. *Algoritmo memético* 83
35. *Comparativa de complejidad computacional de los algoritmos analizados* 84
36. *Análisis asintótico de dias2014* 85
37. *Análisis asintótico de brito2019* 87
38. *Análisis asintótico de soncco2013* 88
39. *Análisis asintótico de al2014* 91
40. *Análisis asintótico de soncco2014* 94
41. *Análisis asintótico de dias2014 vs. brito2019 vs. propuesta* 99
42. *Análisis asintótico de soncco2013 vs. al2014 vs. soncco2014 vs. propuesta* 104

Índice de algoritmos

1. Ventana corrediza - dias2014 - parte 1 59
2. Ventana corrediza - dias2014 - parte 2 60
3. Busqueda anticipada - brito2019 63
4. Algoritmo genetico con eliminación de puntos de interrupción - soncco2013 65
5. Optimización por colonia de hormigas mejorado - al2014 68
6. Algoritmo memético - soncco2014 70
7. Propuesta 1 - Reversión de prefijo de doble lado 98
8. Propuesta 2 - Bafna - Pevzner Metaheurístico 101

Introducción

Capítulo 1

1. INTRODUCCIÓN

El problema de ordenación de permutaciones por inversión de prefijos (SBPR) es fundamental en biología computacional y bioinformática. Consiste en reorganizar una permutación dada invirtiendo secuencias contiguas de sus elementos hasta obtener la permutación identidad. Este problema reviste gran importancia en el estudio de la reordenación del genoma, la evolución y la filogenia, así como en el diseño y optimización de protocolos de secuenciación del ADN. Sin embargo, dada su naturaleza combinatoria, encontrar la solución óptima para el problema SBPR es NP-difícil, lo que impulsa la creación de algoritmos heurísticos y metaheurísticos eficientes. En esta sección se presenta una breve introducción al problema, sus antecedentes, algunas de sus aplicaciones y los objetivos de este estudio.

1.1. Planteamiento del problema

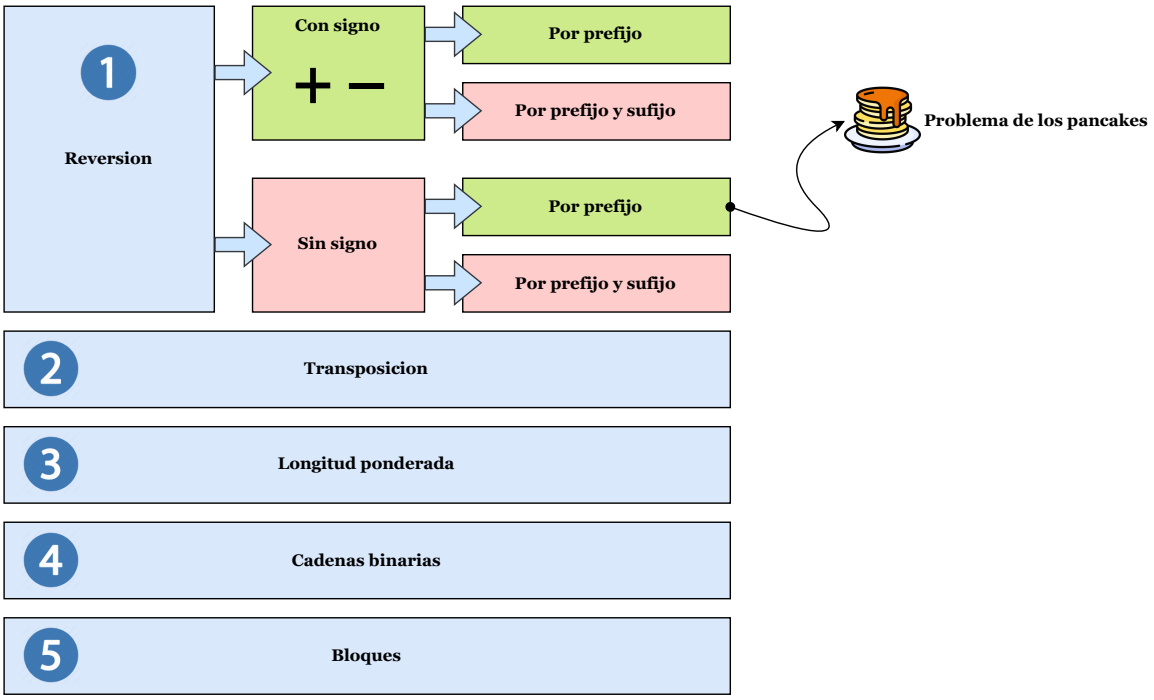
Las permutaciones son el modelo matemático principal en el estudio del ordenamiento cromosómico en Biología Molecular. En particular, los algoritmos de ordenamiento han demostrado tener una estrecha relación con la comprensión de la evolución y cómo los organismos mutan entre sí. En este contexto, los estudios realizados por Bafna y Pevzner (1996) han sido fundamentales, iniciando una línea de investigación que perdura hasta hoy. De hecho, en la literatura se encuentran diversos enfoques, desde la estimación de cotas mediante métodos exactos hasta el desarrollo de algoritmos heurísticos. Sin embargo, las técnicas metaheurísticas han sido poco exploradas, a pesar de ser soluciones potencialmente más efectivas en términos computacionales.

Por otro lado, existe una familia bien definida de problemas relacionados con el ordenamiento de permutaciones. Es relevante mencionar que Lintzmayer (2016) realizó una clasificación exhaustiva en torno a este conjunto de problemas.

Como muestra la figura 1, para la elaboración de esta tesis se tomó el caso de ordenamiento por reversión de prefijo sin signo o SBPR, más comúnmente conocido el problema de los panqueques. Es considerado como uno de los más importantes en el campo de la Genética Computacional.

Figura 1

Familia de problemas sobre ordenamiento de permutaciones



Nota. En esta imagen se muestra una clasificación del problema general de reversión y otros problemas similares. Fuente: Elaboración propia.

Sin embargo, el problema tiene una complejidad de orden factorial (equiparable al problema del agente viajero o al problema de la mochila), con lo cual se hace realmente intratable mediante métodos exactos. Una buena alternativa podría ser el uso de algoritmos heurísticos, cuyo tiempo de ejecución suele ser rápido (esto es porque han sido diseñados para resolver problemas concretos), pero sus soluciones no siempre son muy buenas. Por otro lado, los algoritmos metaheurísticos son mucho más generales y el costo computacional que se debe pagar por la calidad de las soluciones que ofrece se considera razonable. De hecho, Bulteau et al. (2012) demostraron que el problema pertenece a la clase de complejidad NP-Hard, esto brinda un indicador clave para la implementación de una metodología que brinde la flexibilidad y potencia de los algoritmos metaheurísticos.

Los estudios de Rudolph (1994) y Yang y He (2019) muestran cómo la convergencia de estos métodos es posible ya que, por ejemplo, en el caso de los algoritmos genéticos al introducir probabilidades en los operadores de mutación y selección variables en

el tiempo, las posibilidades de convergencia al óptimo aumentan drásticamente. Por supuesto, esto dependerá de la profundidad con la que se analice cada algoritmo y de las principales ventajas que se puedan obtener.

Con todo esto, la idea central de esta tesis está relacionada con el diseño e implementación de algoritmos heurísticos y metaheurísticos que requieren espacios de búsqueda mucho más amplios y exhaustivos. En concreto, los algoritmos heurísticos a implementar son: ventana corrediza y búsqueda anticipada. Por otro lado, las implementaciones metaheurísticas son las siguientes: genético con eliminación de puntos de interrupción, memético basado en GA y de colonias de hormigas.

En la actualidad, la literatura científica carece de estudios en idioma español que aborden este tipo de problemas junto con sus soluciones. Por esta razón, es importante el desarrollo de nuevos aportes que ayuden a fortalecer los conocimientos sobre esta línea de investigación y al mismo tiempo motiven a nuevos investigadores de habla hispana a realizar sus contribuciones basadas en esta primera experiencia.

1.2. Antecedentes

Se dice que hay tres tipos de matemáticos: el que resuelve problemas, el que los propone y el que hace ambas cosas. Un ejemplo claro de esto fue Pierre de Fermat, quien usualmente intercambiaba correspondencia con sus contemporáneos como Pascal y Descartes sobre los problemas más populares de la época.

Con el paso de los años esta práctica continuó entre la comunidad matemática, hasta que muchas revistas especializadas decidieron tomar un mayor protagonismo de este proceso, especialmente desde el siglo pasado donde la ciencia estuvo en el punto más álgido de toda su historia. En este contexto academicista es donde se origina este famoso problema.

Todo inició con Dweighter, (1975) y su publicación en la prestigiosa revista American Mathematical Montly sobre un problema[1] con el siguiente enunciado (traducido del inglés):

[1]Por su alusión realizada a los panqueques, también es conocido como el problema de los panqueques (en inglés, pancake problem) o SBPR.

> El chef de nuestro local es descuidado y cuando prepara una pila de panqueques, salen de diferentes tamaños. Antes de entregarlos al cliente, de camino a la mesa, los reorganiza (de modo que el más pequeño se ubique arriba, y así sucesivamente, hasta que el más grande esté abajo) tomando varios de la parte superior y volteándolos por encima. Repitiendo esto (es decir, variando el número de vueltas) tantas veces como sea necesario. Si hay n panqueques, ¿cuál es el número máximo de vueltas (en función de n) que tendré que usar para ordenarlos? (Dweighter, 1975, p. 1010).

Es importante aclarar que no se trata de un problema de ordenamiento en el sentido estricto, puesto que la operación de invertir o dar vuelta a una cierta cantidad de elementos ya está definida. Esto significa que el objetivo real es encontrar una secuencia de ordenamiento que permita realizar las respectivas inversiones a la permutación dada.

Un par de años después Dweighter et al. (1977) mostraron una secuencia de n para el caso de maximización.

n	1	2	3	4	5	6	7
$f(n)$	0	1	3	4	5	7	8

Siendo:

$$n + 1 \leq f(n) \leq 2n - 6, \forall n \geq 7.$$

Es importante señalar que el planteamiento original sugiere resolver el caso de maximización. Por otro lado, Gates y Papadimitriou (1979) mostraron que es posible ordenar una permutación en $\frac{5n}{3}$ inversiones. Adicionalmente, propusieron una nueva variante del problema utilizando signos negativos y positivos para cada elemento, esto también conocido como problema de los panqueques quemados.

De hecho establecieron cotas razonables con el objetivo de acercarse sutilmente a la solución óptima:

$$f(n) \leq \frac{(5n+5)}{3}$$

Figura 2

Bill Gates, fundador de Microsoft

Nota. Bounds for sorting by prefix reversal es el único artículo académico en el que aparece Bill Gates antes de fundar Microsoft en ese mismo año. Tomado de Meijer (2020).

$$f(n) \geq \frac{17n}{16}, \forall n \to 16n.$$

A pesar del remarcado interés de muchos investigadores en problemas combinatorios por su aplicabilidad inmediata en escenarios reales, la siguiente década se caracterizó por una notable inactividad académica y contribuciones menores. Es posible que durante esa época el problema se haya sido visualizado como un reto o puzzle en lugar de un verdadero problema de investigación.

Muchos años después ocurrió lo que para muchos fue el principal punto de inflexión de esta historia. A finales de siglo pasado, Bafna y Pevzner (1996) encontraron una especial relación entre el ordenamiento genómico y cómo este caso particular del problema puede modelar su comportamiento de una forma natural.

Observaron principalmente cómo la distancia entre dos permutaciones aleatorias está relacionada en cierto sentido con su diámetro[2]. Sin embargo, es importante aclarar que previo a este estudio hubieron muchos otros, pero quizás este tuvo una mayor visibilidad y aceptación entre la comunidad.

[2]Véanse las definiciones más adelante en el capítulo 3.

Como es evidente, a partir de ese momento empezaron a surgir estudios más completos que van desde algoritmos de aproximación hasta la implementación de metaheurísticas que proponían distintos enfoques para encontrar esa secuencia de ordenamiento óptima requerida.

En esta misma línea, Fischer y Ginzinger (2005) construyeron un poderoso algoritmo que representa la primera aproximación polinomial con un radio de 2 unidades. Probablemente el enorme éxito de esta propuesta se basa en la introducción de dos nuevos conceptos en la literatura: puntos de interrupción y grafo de puntos de interrupción. Más adelante en el capítulo 3 se abordarán con mayor detalle sus usos e importancia.

Posteriormente, Zhongxi y Tao (2006) idearon una de las primeras propuestas metaheurísticas aplicadas a problemas de ordenamiento de genoma. Aunque en realidad hicieron énfasis en la creación de un modelo intente resolver el caso de las permutaciones con signo, muy similar al problema original pero, según ellos, ligeramente más simple. Mostraron mejoras significativas en los tiempos de ejecución en comparación a los resultados obtenidos por Christie (1998), quien propuso un algoritmo de aproximación para el caso general del problema[3].

Dos años después Sharmin et al. (2008) desarrollaron una propuesta de aproximación en el cual mostraron algunas mejoras en su radio con $(3 - \frac{3r}{b(\pi)+r})$, sin embargo, este fue uno de los tantos intentos por encontrar una solución funcional pero sin grandes resultados significativos.

A pesar que el algoritmo dominante seguía siendo el de Fischer y su equipo, Chitturi et al. (2009) desarrollaron una exposición matemática sobre la posibilidad de transformar la idea original de Gates y Papadimitriou (1979) en una serie de casos que pudiera converger hacia una cota superior de $\frac{18}{11}n$. De hecho, este planteamiento ha sido aceptado por la comunidad hasta el día de hoy.

Por otro lado, Helmert (2010) desarrolló una propuesta novedosa, aunque injustamente ignorada por muchos autores llamada heurística de brecha, especialmente diseñado para problemas de planeación y horarios. En realidad, no se brindan muchos detalles sobre su implementación pero sí sobre los resultados de su funcionamiento.

[3]El caso general del problema es conocido como Sorting By Reversals (SBR). En esta tesis, se aborda solo uno de los casos particulares: Sorting By Prefix Reversals (SBPR).

Algunos años después ocurre lo que para muchos es considerado el segundo punto de inflexión en la historia del problema: Bulteau et al. (2012) demostraron que el problema es NP-Hard. Esto evidentemente planteó un cambio de mentalidad en la comunidad y a partir de este momento muchos de los esfuerzos estuvieron enfocados en el desarrollo de soluciones heurísticas y metaheurísticas que permitan hacer más manejables grandes instancias del problema y aún así obtener muy buenos resultados.

En este sentido, Soncco-Álvarez y Ayala-Rincón (2012) desarrollaron la primera propuesta de algoritmos genéticos conocida después de la demostración. El principal reto en este punto fue modificar el flujo natural del algoritmo genético para intentar adaptarlo al problema. Su impacto fue significativo ya que mejoró en promedio los resultados de Christie (1998). Otra cosa que es importante resaltar es que han sido los primeros autores en publicar su implementación en un repositorio público[4].

Es crucial resaltar además el estudio realizado en el marco de la tesis doctoral llevada a cabo por Bulteau (2013) en donde explican más a fondo los conceptos de base utilizados para demostrar que el problema es NP-Hard. Todas estos son ideas extraídas de campos como el álgebra lineal, la teoría de conjuntos y el análisis combinatorio.

Ese mismo año Soncco-Álvarez y Ayala-Rincón (2013) mejoraron su idea anterior de tal manera que concibieron un método sistemático para eliminar *puntos de interrupción* aunque esta vez analizando más de cerca a las permutaciones con signo. De hecho, este grupo de autores han desarrollado el trabajo más significativo en materia de algoritmos genéticos en los últimos 20 años.

Aunque los algoritmos basados en fenómenos genéticos gozan de una enorme popularidad por su sencillo mecanismo de convergencia y facilidad de implementación, es importante mencionar que existen alternativas muy competitivas como los basados en SI (en español, inteligencia de enjambres) tales como: hormigas, abejas, aves, murciélagos, libélulas y demás, cuyos procedimientos se han estandarizado con el paso del tiempo. Con esta base, Al Daoud (2014) propuso una solución basada en una colonia de hormigas que en su defecto necesitan encontrar la secuencia de ordenamiento óptima haciendo uso nuevamente del concepto de puntos

[4]Cuando se realizan trabajos de investigación en la línea del desarrollo de algoritmos es vital hacer que el código de implementación sea visible y evaluable por otros. En este artículo se explica como hacerlo. https://guides.lib.berkeley.edu/citeyourcode

de interrupción. Sus resultados, aunque no muy populares se consideran pioneros por su originalidad.

Ese mismo año Soncco-Álvarez y Ayala-Rincón (2014) presentaron su nuevo avance en materia de algoritmos meméticos, esta vez haciendo un verdadero énfasis al tema de estudio de esta tesis. Posiblemente esta sea la única propuesta metaheurística directa que se haya aplicado al problema en toda su literatura. De hecho, este enfoque demostró tener resultados más competitivos en comparación con el algoritmo genético estándar e inclusive su versión implementada utilizando computación paralela.

De forma similar, los siguientes autores han propuesto un verdadero corpus de soluciones heurísticas de las más variadas. En el caso de U. Dias et al. (2014) diseñaron un marco de trabajo holístico para aproximar soluciones en familias completas de problemas como el de las permutaciones por reversión, llamaron a su algoritmo principal Ventana Corrediza debido a que inicialmente se tiene una estructura de datos que itera y mejora su aproximación hasta el final de la ejecución, sigue una filosofía muy parecida a un algoritmo voraz[5].

Dentro de este marco, Z. Dias y Dias (2015) presentaron un nuevo punto de vista relacionado con el ordenamiento por inversiones y transposiciones. A pesar que la relación con el problema de estudio no es muy fuerte, sigue siendo una fuente indirecta de ideas para plantear soluciones con un fundamento más profundo. En este caso, lograron mejorar asintóticamente el factor de aproximación de 3 a 2 unidades.

De hecho, en este grupo de investigadores surgió el gran trabajo de tesis doctoral de Lintzmayer (2016) quien se dio a la tarea de formalizar la notación matemática y a su vez estudiar 45 casos específicos de ordenamiento del genoma (entre ellos, el problema de estudio). Adicionalmente, desarrolló una completa composición de algoritmos que mejoraban muchas de las aproximaciones conocidas además de mostrar un nuevo enfoque en el planteamiento de permutaciones con longitud ponderada.

Siempre dentro del mismo círculo de investigadores, Brito et al. (2019) tomaron como base la propuesta anterior de U. Dias et al. (2014) desarrollando de esta forma una nueva versión de la ventana corrediza y dos nuevas propuestas: búsqueda anticipada y ventana corrediza iterativa. Demostraron que, el primero de estos puede ejecutarse en tiempo lineal a diferencia de la versión anterior y los demás

[5]Es una forma de encontrar una solución óptima local en cada paso con alguna probabilidad de llegar a la solución óptima general.

también fueron diseñados para brindar soluciones de calidad para instancias muy grandes.

Es muy importante aclarar que la literatura relacionada con este problema es relativamente corta. Se tomaron en cuenta artículos que hacían énfasis tanto en el caso general y como en algunos casos particulares por su valioso aporte teórico y que al mismo son considerados por la comunidad como los estudios más relevantes dentro de toda esta nueva disciplina.

1.3. Justificación

El 31 de marzo de 2022, la revista Science anunció un hito histórico en la ciencia moderna: la secuenciación completa del genoma humano por primera vez. Nurk et al. (2022) y un equipo de más de 91 investigadores confirmaron que desde el año 2000 se ha explorado el 8 % restante del genoma (para un análisis más completo ver figura 3). Este logro es el resultado de un esfuerzo colectivo y multidisciplinario que involucra diversas áreas del conocimiento, destacando el estudio de algoritmos eficientes para la transposición de elementos en el ADN (ácido desoxirribonucleico) repetitivo.

Es interesante observar cómo en este estudio se utilizaron al menos seis enfoques distintos sobre ordenamiento y algoritmos de secuenciación, a pesar de su carácter técnico desde el punto de vista biológico. Sin embargo, aquí es necesario destacar la importancia del desarrollo de trabajos como este, ya que sirven como ciencia de apoyo para contribuciones mucho más grandes y significativas que ayudan a comprender más de cerca el verdadero funcionamiento de la naturaleza.

Aunque las aplicaciones de este estudio van más allá del ámbito de la biología molecular, sin duda esta es una de las más relevantes. Es ampliamente utilizada en la investigación de enfermedades como la hemofilia tipo A, así como en el estudio de la evolución de nuevas especies.

Estas son algunas implicaciones prácticas relacionadas con el problema de estudio:

- **Ordenamiento no estándar de datos**: Las estructuras de datos clásicas de las ciencias computacionales permiten ordenar enormes cantidades de datos en tiempo polinomial sin ningún problema, gracias a algoritmos como QuickSort, MergeSort y HeapSort. Sin embargo, SBPR ofrece una nueva perspectiva sobre

el comportamiento e implementación de estos algoritmos, ya que se considera un híbrido entre un algoritmo de ordenamiento y uno de búsqueda (de Cabezón Irigaray, 2020).

- **Procesamiento paralelo**: Aunque el procesamiento paralelo es una técnica establecida en la práctica, las industrias dedicadas al diseño y construcción de microprocesadores siempre buscan el mejor algoritmo para maximizar la velocidad de procesamiento. La literatura ha demostrado que, en general, SBPR es un excelente candidato para modelar situaciones como esta y proporcionar soluciones razonablemente buenas. Estudios como los de Soncco-Álvarez et al., 2013 y L. A. da Silveira et al., 2018 han demostrado que esto es posible, siempre y cuando se reproduzcan las condiciones ambientales mínimas adecuadas.

- **Ordenamiento del genoma**: El problema SBPR es fundamental en el campo del ordenamiento genómico, desempeñando un papel crucial en la evolución y el descubrimiento de nuevas enfermedades. Al resolver el problema SBPR, los investigadores pueden identificar la secuencia de acontecimientos que condujeron a la formación de un genoma determinado y comprender cómo han evolucionado los genomas a lo largo del tiempo. Varios estudios han utilizado el problema SBPR para estudiar la evolución de los genomas, incluido el genoma humano y el genoma bacteriano.

- **Reconstrucción filogenética**: El problema SBPR también se utiliza en el campo de la reconstrucción filogenética, que implica inferir la historia evolutiva de especies o genes. Resolviendo el problema SBPR, los investigadores pueden construir un árbol filogenético que represente las relaciones evolutivas entre diferentes especies o genes. Varios estudios han utilizado el problema SBPR para reconstruir el árbol filogenético de varios organismos, incluyendo bacterias y plantas.

- **Secuenciación del ADN**: El problema SBPR también es relevante para el diseño y la optimización de protocolos de secuenciación de ADN. Al resolver el problema SBPR, los investigadores pueden identificar la forma más eficiente de secuenciar un genoma o gen determinado. Varios estudios han utilizado el problema SBPR para optimizar protocolos de secuenciación de ADN, incluyendo la secuenciación de genomas bacterianos y ADN mitocondrial.

Por otro lado, el uso de heurísticas y metaheurísticas siempre ha sido un recurso experimental valioso para descubrir qué algoritmos funcionan mejor para casos específicos y variantes del problema. En cualquier caso, el principal aporte será mostrar una familia de algoritmos de este tipo que mejoren las soluciones actuales propuestas en el estado del arte.

Finalmente, después de comparar y poner a prueba la eficiencia de dichos algoritmos, se seleccionarán los mejores como referencia para diseñar una propuesta unificada que brinde los mismos o mejores resultados obtenidos previamente. Es decir, el aporte metodológico tiene un carácter experimental, ya que se han explorado diversas alternativas para solucionar el problema.

1.4. Área del conocimiento

Según el Reglamento del Sistema de Estudios de Posgrado y Educación Continua de La Universidad Nacional Autónoma de Nicaragua, UNAN Managua, este estudio está clasificado en el área de Ciencias Aplicadas, siguiendo la línea de investigación de Modelación y Programación Matemática, debido a que se ha desarrollado un marco teórico en donde se han expuesto de principales propiedades de los modelos, métodos, técnicas matemáticas de optimización combinatoria y todas las posibles soluciones que la literatura especializada ofrece, todo esto integrado con herramientas tecnológicas que han permitido realizar un análisis detallado con los datos producidos en los procedimientos estudiados.

1.5. Tema y objetivos de investigación

1.5.1. Tema de investigación

Diseño de algoritmos heurísticos y metaheurísticos eficientes para el problema SBPR (Sorting Permutations By Prefix Reversals).

1.5.2. Objetivo General

- Diseñar algoritmos heurísticos y metaheurísticos eficientes para el problema SBPR.

1.5.3. Objetivos Específicos

1. Describir la teoría matemática base para el estudio, diseño e implementación de algoritmos heurísticos y metaheurísticos para el problema SBPR.
2. Analizar el diseño y estructura de los algoritmos heurísticos y metaheurísticos para el problema SBPR.
3. Implementar algoritmos heurísticos y metaheurísticos existentes y propuestos utilizando el lenguaje de programación Python para el problema SBPR.
4. Comparar los resultados obtenidos de los algoritmos heurísticos y metaheurísticos para el problema SBPR mediante experimentación computacional.

1.6. Contribuciones

Este estudio de investigación, con una duración de aproximadamente 27 meses de esfuerzo continuo, ha generado contribuciones significativas al campo de conocimiento en cuestión, tal como se detalla en la sección 1.7 y capítulo 6.

A continuación se listan las principales contribuciones de esta tesis doctoral en la línea de investigación Modelación y Programación Matemática:

- **Identificación de la complejidad del problema:** La tesis destaca que el problema SBPR ha sido mostrado como un problema NP-Hard desde 2012, lo que significa que encontrar una solución óptima en tiempo polinómico o inferior es inviable en términos prácticos.

- **Desarrollo de algoritmos heurísticos y metaheurísticos:** Durante la investigación se desarrollaron e implementaron algoritmos heurísticos y metaheurísticos basados en criterios de la literatura revisada. Estos algoritmos buscan proporcionar soluciones aproximadas eficientes para el SBPR, superando las limitaciones de encontrar soluciones óptimas.

- **Propuesta de una familia de metaheurísticas inspiradas en la vida:** Se introdujo una nueva familia de metaheurísticas inspiradas en la vida, las cuales han demostrado ser eficientes en la búsqueda de soluciones en grandes espacios de búsqueda. Este enfoque innovador puede representar una contribución significativa al campo de la optimización combinatoria.

- **Implementación práctica con Python y Google Colab:** Se destaca la implementación de los algoritmos utilizando el lenguaje de programación Python y la herramienta Google Colab. Esto no solo presenta una aplicación práctica de los métodos propuestos, sino que también proporciona una plataforma accesible para la comunidad académica y científica.

- **Revisión de conceptos básicos y métodos de ordenación:** La tesis incluye una revisión exhaustiva de los conceptos básicos relacionados con la teoría moderna de permutaciones y sus métodos elementales de ordenación. Esto contribuye a establecer el contexto teórico necesario para comprender y abordar el SBPR.

- **Evaluación y comparación de resultados:** La investigación evaluó los algoritmos utilizando conjuntos de resultados proporcionados por la literatura científica de los últimos 10 años. Se analizaron las ventajas y desventajas de los diferentes métodos propuestos, proporcionando información valiosa sobre su rendimiento relativo.

1.7. Publicaciones y conferencias asistidas

- **Publicaciones**:
  - Incidencia de los estudios sobre ordenamiento genómico en la secuenciación del genoma humano - (2023)
  - Una revisión y clasificación de los algoritmos para el problema SBPR (Sorting Permutations By Prefix Reversals) - (2023)
- **Conferencias**:
  - Diseño de algoritmos heurísticos y metaheurísticos eficientes para el problema SBPR (Sorting Permutations By Prefix Reversals) - Keiser University - (2023)
  - Metaheuristics for sorting permutations by prefix reversals (SBPR) - XXIII International Symposium of Mathematical Methods Applied to Sciences - (2021)

1.8. Organización de la tesis

La presente investigación está estructurada en cinco capítulos. El capítulo 1 corresponde a la introducción en donde se abordan aspectos generales y metodológicos tales como el planteamiento del problema, los antecedentes, la justificación y los objetivos de tesis. En estos apartados se explica la importancia del problema y su enorme impacto aplicativo principalmente a las áreas relacionadas con Biología Computacional. También se describen los aportes más significativos realizados sobre esta línea de investigación desde que el problema fue planteado por primera vez en 1975. De igual forma se establecen los objetivos específicos como una forma lógica de mostrar cada una de las etapas del proceso metodológico y técnico que se tomaron en cuenta para desarrollar este documento.

Posteriormente en el capítulo 2 correspondiente a la **reseña histórica del problema SBPR o estado del arte** se hace un énfasis especial en los aportes realizados en los últimos diez años de la literatura del problema. Todos estos estudios hacen referencia a algoritmos heurísticos y metaheurísticos.

Luego, en el capítulo 3 se realiza una descripción sobre los principales conceptos de la teoría de permutaciones moderna y cómo estas definiciones ayudan a entender el planteamiento del problema desde su raíz. También se describen brevemente los principales algoritmos heurísticos y metaheurísticos escogidos como base de esta tesis para su posterior análisis.

En el capítulo 4 se describe el funcionamiento de cada uno de estos algoritmos con mayor detalle, tomando en cuenta su descripción en pseudocódigo, sus principales ventajas y sus debilidades en función de su desempeño computacional.

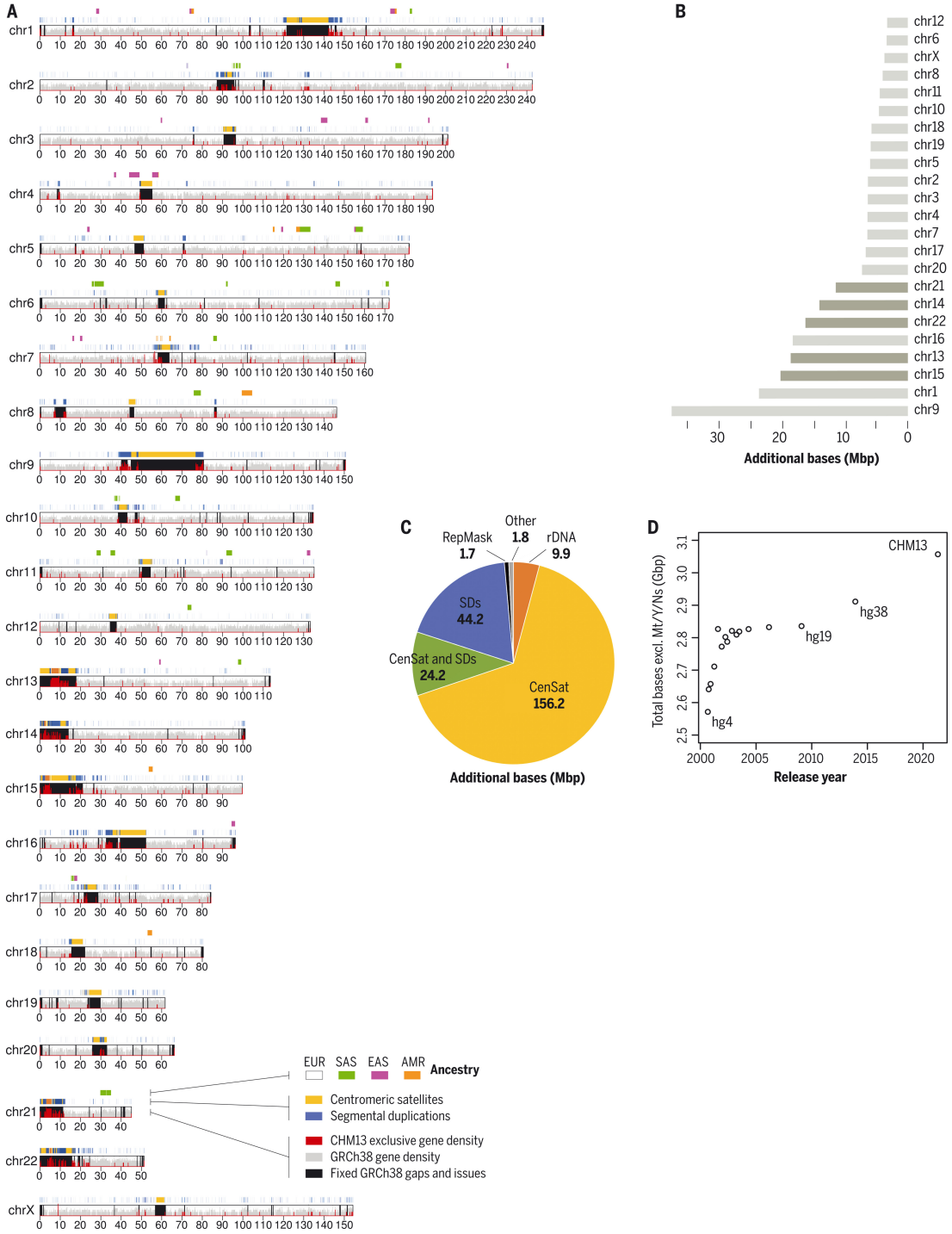
En el capítulo 5, se realiza una descripción de la metodología utilizada para el análisis de los algoritmos estudiados y propuestos, así como las principales herramientas computacionales utilizadas para lograr estos objetivos. Finalmente se brinda una descripción del cronograma del tesis, partiendo desde la idea de investigación hasta la finalización de este documento.

Después en el capitulo 6 se realiza un análisis comparando la eficiencia de los algoritmos analizados anteriormente contra contra su implementación real en el lenguaje de programación Python. Esta se considera la parte más experimental de todo el proceso ya que aquí se van a determinar los algoritmos que mejor se ajustan para solucionar el problema de forma aproximada. Cabe destacar que también se han propuesto un par de algoritmos para mostrar cual de todos estos resulta ser el más eficiente en términos de complejidad computacional y tiempo de ejecución consumido.

Finalmente se presentan una serie de conclusiones relacionadas con todo el proceso de investigación, la bibliografía científica utilizada (en su mayoría en idioma inglés) y los anexos en donde se mostrará parte del código utilizado para implementar el análisis de todos los algoritmos mencionados en este trabajo de tesis.

Figura 3

Resumen del mapeo completo del genoma humano



Nota. Análisis completo del genoma humano.
Tomado de Nurk et al. (2022)

Reseña histórica del problema SBPR

Capítulo 2

2. RESEÑA HISTÓRICA DEL PROBLEMA SBPR

Siempre ha existido un enorme debate acerca de los límites de tiempo que deberían de tener los estudios en cuanto a revisión bibliográfica se refiere. En este sentido Sjøberg (2010) desarrolló uno de los informes más detallados respecto al tema utilizando datos de más de 6 mil revistas del JCR Science Edition y otras fuentes como el Social Science Edition. El estudio llegó a la conclusión que, en promedio, para las Ciencias de la Computación el tiempo solía ser de 7.5 años, mientras que en Matemática y aplicaciones interdisciplinarias la media rondaba los 9.7 y 7.0 años respectivamente.

Aunque, como es evidente, esto no debe considerarse como una regla general, puesto que todas las investigaciones tienen enfoques diferentes y distintos niveles de complejidad. Técnicamente, para la elaboración de esta tesis debería considerarse la media aritmética de los datos anteriores como una medida razonable, es decir, 8.06 años. Sin embargo, dado que la concentración más importante de métodos, técnicas y algoritmos han surgido en los últimos 10 años, entonces se tomó la decisión de ampliar ese límite a 2 años adicionales.

2.1. Algoritmos metaheurísticos

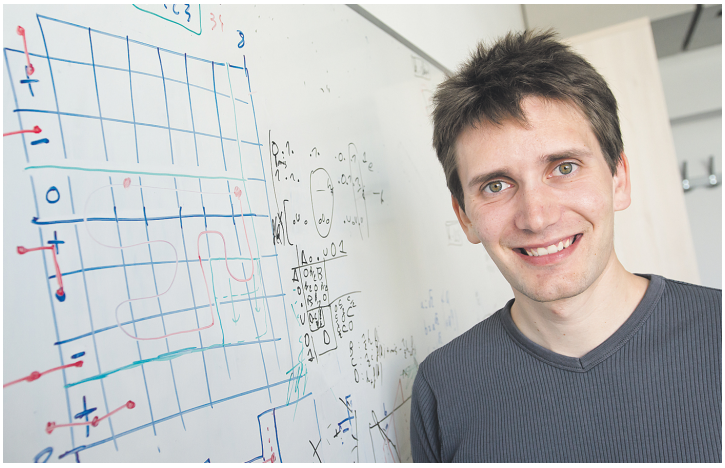
Desde que Bulteau et al. (2012) publicó su demostración sobre SBPR la dinámica para plantear posibles soluciones al problema cambió radicalmente (ver figura 4. Muchos autores optaron por desarrollar soluciones heurísticas, ya que algorítmicamente es la opción que tiene más sentido y en la que se debería invertir un mayor esfuerzo. De esta forma, inició la era de las heurísticas y metaheurísticas para SBPR. [6]

Antes de analizar algunos de los estudios más importantes, hay que aclarar que los trabajos de Zhongxi y Tao (2006), Sharmin et al. (2008) fueron excluidos, en parte por las razones explicadas en el apartado anterior y adicionalmente porque muchos de los trabajos posteriores mejoraron en gran medida sus propuestas. Por tal razón se consideró redundante brindarles un espacio adicional, por el mínimo valor que estos aportan.

[6] Por cohesión, los algoritmos heurísticos deberían analizarse en primera instancia, sin embargo, se optó por utilizar el orden cronológico como herramienta narrativa.

Figura 4

Dr. Laurent Bulteau en 2014



Nota. El Dr. Bulteau demostró que el problema MIN-SBPR es NP-Hard. Tomado de TU Berlin/PR/Ulrich Dahl

En contraste, autores cómo Soncco-Álvarez y Ayala-Rincón (2013) podría considerarse como el más prolífico de todos. En esta ocasión, su aporte está relacionado con un algoritmo genético modificado capaz de eliminar puntos de interrupción, especialmente para permutaciones con signo. Para cualquier caso del problema en cuestión, este enfoque podría resultar efectivo, dado que uno de los objetivos es minimizar los puntos de interrupción a cero lo más rápidamente posible, logrando así la permutación identidad deseada.

Por otra parte, también se han desarrollado métodos alternativos en contraste con los algoritmos genéticos, como aquellos fundamentados en SI (hormigas, abejas, aves, murciélagos, libélulas, entre otros) cuyos procedimientos se han estandarizado con el paso del tiempo. Ante esto, Al Daoud (2014) propuso una solución mejorada con base en la optimización por colonia de hormigas que en su defecto necesita encontrar la secuencia de ordenamiento óptima haciendo uso nuevamente del concepto de puntos de interrupción. Este método propone el uso de la función de evaporación de feromonas, propuesta inicialmente por Dorigo et al. (1996).

Sin embargo, en cuanto a las implementaciones genéticas, Soncco-Álvarez y Ayala-Rincón (2014) presentaron una novedosa variación referente a algoritmos meméticos, esta vez haciendo un verdadero énfasis al tema de estudio de esta tesis. De hecho, esta

sea la única propuesta metaheurística directa que estudiará a fondo sobre el problema durante el desarrollo y análisis de algoritmos mediante programación. Es importante aclarar que este enfoque obtuvo mejores resultados en comparación con un algoritmo genético estándar e inclusive a su versión propia del autor utilizando computación paralela.

Como consecuencia, dicho estudio, desarrollado por L. Â. da Silveira et al. (2017) fue excluido nuevamente debido a su redundancia, y principalmente por las dificultades técnicas que presentaba. Esto se debe a que las especificaciones del algoritmo requieren una red distribuida de computadoras potentes con altos requisitos de hardware. Sin embargo, este trabajo se limita al uso del plan gratuito de Google Colab, el cual restringe el poder de cómputo y no permite configuraciones avanzadas. [7]

2.2. Algoritmos heurísticos

Suele afirmarse que los algoritmos metaheurísticos son más poderosos porque intentan cubrir y *esquivar* esos *óptimos locales* que muchas veces suelen ser problemáticos. Sin embargo, una técnica heurística correctamente planteada también suele ser muy poderosa.

En ocasiones, estas propuestas pueden presentar muchos puntos de mejora en cuanto a su capacidad de convergencia e implementación computacional. Tal fue el caso de Caprara et al. (1999) quienes desarrollaron su trabajo basados en una solución clásica dentro de la optimización combinatoria como es la generación de columnas utilizando branch-and-bound. A pesar de su enorme popularidad y efectividad, faltó muy poco para ser destronada por técnicas mucho más modernas.

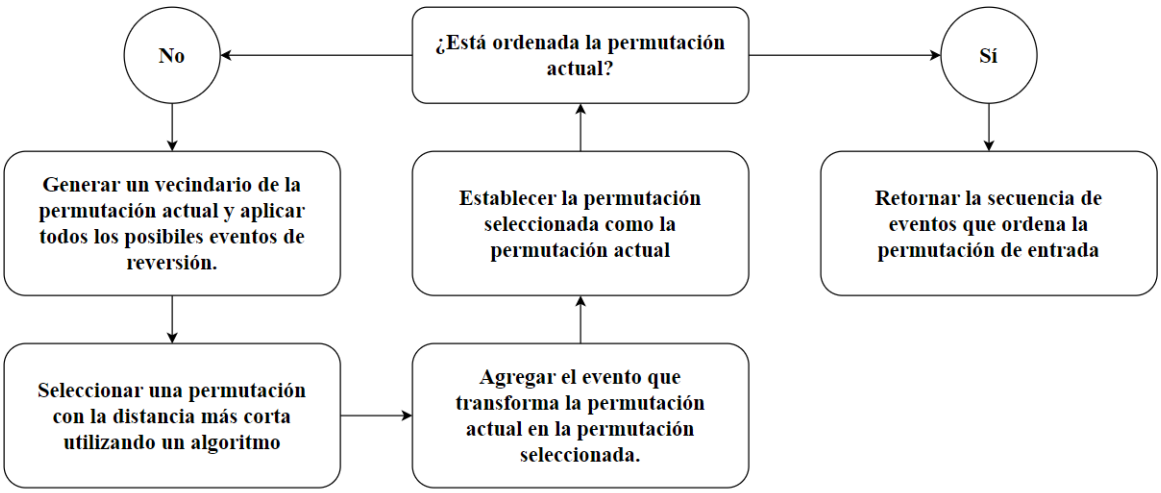
Como respuesta a lo anterior, Helmert (2010) propuso el algoritmo de punto de referencia basado en conceptos provenientes de la AI (en español, inteligencia artificial) y el ML (en español, aprendizaje automático). Aunque, por desgracia, ha sido complicado encontrar el artículo original con sus novedosos resultados, debido a que la única versión que existe está incompleta. Fueron estas razones las que llevaron nuevamente a excluir ambos estudios tanto de su análisis como de su implementación.

[7] Puede leer más sobre distintos planes que ofrece Google Colab en el siguiente enlace: https://colab.research.google.com/signup

Por su parte, U. Dias et al. (2014) diseñaron un marco de trabajo completo que consiste en aproximar soluciones para varios de los casos más populares de problemas de ordenamiento genómico entre ellos el SBPR. Ellos mismos llamaron a su algoritmo principal ventana corrediza (ver figura 5) debido a que se trata de un método de mejora continua, siguiendo una filosofía muy similar a un algoritmo greedy o voraz.

Figura 5

Diagrama de flujo de ventana corrediza



Nota. Diagrama de flujo que muestra el proceso algorítmico de una ventana corrediza. Tomado de Brito et al. (2019)

De forma similar, Brito et al. (2019) tomaron como base la propuesta y desarrollaron una nueva versión de la ventana corrediza junto a dos propuestas adicionales: Mirada hacia el futuro y Ventana Corrediza Iterativa. Entre otras cosas mostraron que, el primero de estos puede ejecutarse en tiempo lineal a diferencia de su primera versión. Además los otros también fueron diseñados para muy buenas soluciones con instancias del problema los suficientemente grandes (permutaciones con más de 500 elementos).

Hasta la fecha de elaboración del presente trabajo no se tiene conocimiento de nuevas contribuciones y estudios relacionados el tema de esta investigación. Por tal razón, se considera 2019 como el último año de actividad académica, retomado nuevamente por el presente trabajo.

De esta forma, que en la tabla 1 se muestra a modo de resumen el conjunto de todos los algoritmos a implementar en el capítulo 6 y cuya importancia ha tenido un impacto significativo en el estado del arte del problema.

Tabla 1

Algoritmos a implementar en esta tesis

Autor y año	**Tipo de algoritmo**	**Cod. Algoritmo**
Soncco-Álvarez y Ayala-Rincón, 2013	MHA.	`soncco2013`
Al Daoud, 2014	MHA.	`al2014`
Soncco-Álvarez y Ayala-Rincón, 2014	MHA.	`soncco2014`
U. Dias et al., 2014	HA.	`dias2014`
Brito et al., 2019	HA.	`brito2019`

Nota. Esta tabla muestra los diferentes algoritmos que se analizarán e implementarán en los próximos capítulos.

Fundamentos matemáticos del problema SBPR

Capítulo 3

3. FUNDAMENTOS MATEMÁTICOS DEL PROBLEMA SBPR

En este capítulo se realiza un recorrido por los principales conceptos matemáticos desarrollados para estudiar al problema SBPR durante más de cuatro décadas. En su defecto, se utilizan conceptos clásicos como el de permutación extraído de la teoría combinatorio o términos más modernos como el puntos de interrupción, el cual se ha considerado como la idea clave para desarrollar algoritmos eficientes que puedan resolver este problema en un tiempo razonable.

3.1. Permutaciones

Desde la teoría de probabilidad hasta aplicaciones en bioinformática, las permutaciones se han caracterizado por ser un objeto matemático sumamente versátil para modelar muchos fenómenos de la naturaleza. Aunque su conceptualización matemática suele ser sencilla, a menudo, su verdadero poder radica cuando se une con otras ideas, conceptos o áreas de la ciencia.

La definición formal está basada en la teoría de conjuntos. Ante esto, Bóna (2012) define una permutación como:

Definición 1

Una ordenación lineal de los elementos del conjunto $[n] = \{1, 2, 3, ..., n\}$ se denomina permutación, o, si queremos destacar el hecho de que consta de n entradas, una $n - permutación$
(Bóna, 2012, p. 1).

Es decir, es una lista de elementos con un orden único. Cada uno de estos elementos compone al conjunto de esta permutación.

Ejemplo 1

Si $n = 3$, el conjunto con las permutaciones resultantes sería
$123, 132, 213, 231, 312, 321.$

De igual forma, el mismo autor recuerda uno de los hechos más populares sobre las permutaciones en la teoría combinatoria clásica.

Definición 2

El número de $n - permutaciones$ es $n!$
(Bóna, 2012, p. 1).

Esto significa que un conjunto de permutaciones $p = p_1p_2...p_n$ es posible obtener n entradas para p_1, luego $n-1$ entradas para p_2 y así sucesivamente.

Sin embargo, Fertin et al. (2009) sugieren que las *permutaciones lineales* son funciones, entendidas como ordenaciones de un conjunto finito. También aquí se muestra la notación clásica utilizada en combinatoria para denotar una permutación π con dos filas:

$$\begin{pmatrix} 1 & 2 & ... & n \\ \pi_1 & \pi_2 & ... & \pi_n \end{pmatrix}$$

Aunque la notación más común que se puede encontrar en la literatura es:

$$\pi = (\pi_1\pi_2...\pi_n)$$

En el siguiente apartado se muestran las definiciones formales de la teoría matemática que forman la base del problema. La gran mayoría de esta ideas han sido formuladas gracias al trabajo previo realizado por Lintzmayer (2016) y Soncco-Álvarez y Ayala-Rincón (2013).

Definición 3

Permutación: Una permutación π es una biyección de $\{1, 2, \cdots, n\}$ sobre sí misma. El grupo de todas las permutaciones de $\{1, 2, \cdots, n\}$ se denota por S_n y escribimos una permutación π en S_n como $\pi = (\pi_1\pi_2 \cdots \pi_n)$ tal que $\pi(i) = \pi_i$ (Soncco-Álvarez y Ayala-Rincón, 2013, p. 122).

Definición 4

Permutación identidad: Sea $\iota = (1\ 2 \cdots n)$. Informalmente llamada con la permutación ordenada
(Lintzmayer, 2016, p. 17).

Definición 5

Conjunto de todas las permutaciones: Sea S_n el conjunto de todas las permutaciones de tamaño n. Note que $\pi, \iota \in S_n$
(Soncco-Álvarez y Ayala-Rincón, 2013, p. 122).

Ejemplo 2

Sea $S_n = \{(123), (132), (213), (231), (312), (321)\}$, por tanto, $\pi = (132)$ y $\rho = (123)$.

3.1.1. Operaciones

Las operación de reversión de prefijo se define como sigue:

Definición 6

Reversión de prefijo: Sea $\rho(j) = \rho(1, j)$, con $1 \leq j \leq n$
(Lintzmayer, 2016, p. 18).

Ejemplo 3

Sea $\pi = (54123)$ al aplicar una reversión de prefijo $\rho(1, 2)$ se obtiene $\pi = (45123)$.

3.1.2. Formulaciones

Las siguientes definiciones forman la base teórica que define el problema SBPR matemáticamente.

Definición 7

Secuencia de ordenamiento: Sea $\sigma = \{\delta_1, \delta_2, \cdots, \delta_n\}$. Esto es aplicado sobre π usando $\rho(j)$ como una operación para obtener ι Note que $\mid \sigma \mid=$ ejecuciones de $\rho(j)$
(Lintzmayer, 2016, p. 19).

Definición 8

Longitud de una reversión de prefijo con σ: Sea $\delta_k = j$ la longitud de $\rho(j)$ (Lintzmayer, 2016, p. 18).

Definición 9

Conjunto de secuencias de ordenamiento dado π: Sea $\varsigma = \{\sigma_1, \sigma_2, \cdots, \sigma_n\}$ como el conjunto de todas las secuencias de ordenamiento para una permutación π
(Lintzmayer, 2016, p. 19).

Definición 10

Distancia de una reversión de prefijo: Sea $d(\pi) = min\{\varsigma \mid \pi$ una permutación dada $\}$
(Lintzmayer, 2016, p. 19).

Ejemplo 4

Sea $\pi = (54123)$, al aplicar una secuencia de ordenamiento, se tiene:

$$\rho = (1,2) \rightarrow \pi = (45123) \rightarrow \sigma_1 = \{2\}$$

$$\rho = (1,5) \rightarrow \pi = (32154) \rightarrow \sigma_1 = \{2,5\}$$

$$\rho = (1,3) \rightarrow \pi = (12354) \rightarrow \sigma_1 = \{2,5,3\}$$

$$\therefore\ \varsigma = \{\sigma_1\} \rightarrow d(\pi) = 3$$

3.1.3. Problema SBPR

La definición del problema SBPR es la siguiente:

Definición 11

Dada una permutación π se debe encontrar $d(\pi) = min\{\varsigma\}$ utilizando operaciones $\rho(1,j)$, de tal forma que $\pi = \iota$
(Lintzmayer, 2016, p. 19).

En este contexto, es necesario plantear un algoritmo que siempre retorne σ mínimas para siempre obtener una $d(\pi)$ mínima.

3.1.4. Puntos de interrupción

Definición 12

Punto de interrupción para una permutación SBPR: Sean un par de elementos consecutivos (π_{i+1}, π_i), hay un punto de interrupción cuando $|\pi_{i+1}, \pi_i| \neq 1$
(Lintzmayer, 2016, p. 21).

En realidad, la cantidad de puntos de interrupción de π es $b(\pi)$.

3.1.5. Trozos

Definición 13

Trozos: Sea un trozo una secuencia $(\pi_i \cdots \pi_j)$ de elementos de π, con $1 \leq i \leq j \leq n$, de tal forma que:

- $i = 1$ o (π_{i-1}, π_i) es un punto de interrupción.
- $j = n$ o (π_j, π_{j+1}) es un punto de interrupción.
- Ningún elemento consecutivo de la secuencia es un punto de interrupción (Lintzmayer, 2016, p. 22).

Ejemplo 5

Sea $\pi = (54123)$. Los trozos son bloques consecutivos, es decir, para π serían: $(\underline{54}\ \underline{123})$, el punto de interrupción está entre $\underline{4}\ \underline{1}$ y por tanto $b(\pi) = 1$.

3.1.6. Grafos con puntos de interrupción

Definición 14

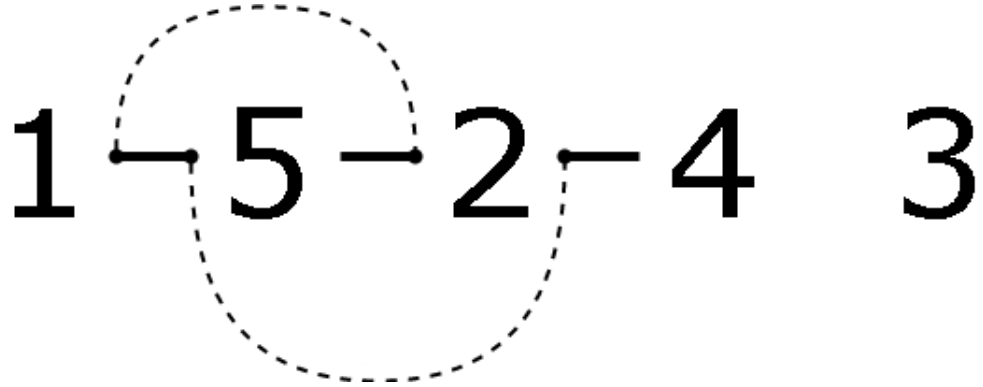
Dada una permutación π, un grafo con puntos de interrupción $G(\pi)$ es un grafo indirecto con aristas coloreadas derivadas de las relaciones de puntos de interrupción en π con los $n+1$ vértices etiquetados como $1, 2, ..., n, n+1$. Los criterios son:

- Dos vértices i y j están unidos con una **arista negra** si (i, j) es un punto de interrupción de π.
- Dos vértices i y j están unidos con una **arista gris** si (i, j) no son consecutivos en π.
  (Lintzmayer, 2016, p. 123).

En la figura 6 se puede obsevar un ejemplo del funcionamiento del concepto anterior. Para $\pi = (15243)$ se puede construir un grafo mapeando los puntos de interrupción y la distancia asociada. De esta forma se pueden plantear algoritmos que opten por elegir caminos más cortos con una configuración similar.

Figura 6

Grafo de puntos de interrupción



Nota. Este es un ejemplo de un grafo con puntos de interrupción, concepto clave que ha permitido desarrollar algoritmos robustos para resolver el problema SBPR. Fuente: Elaboración propia.

3.2. Algoritmos heurísticos

Comúnmente la palabra heurística está relacionada con algo que no precisa ser exacto. Inclusive, hay quienes afirmaron en el pasado que lo heurístico era contrario a lo algorítmico. Hoy en día el término es utilizado para describir un proceso de descubrimiento continuo con base en una experiencia previa.

Desde el punto de vista etimológico, proviene de la raíz griega *heurískein* que significa hallar o inventar. Coloquialmente se podría interpretar como ir más allá, investigar a fondo y descubrir algo nuevo.

A continuación se muestran los procedimientos heurísticos más destacados de esta línea de investigación[8].

3.2.1. Algoritmo de ventana corrediza

Es una técnica utilizada en programación para recorrer iterativamente un arreglo con n elementos, subdiviéndolo en bloques de tamaño k, donde $k \leq n$.

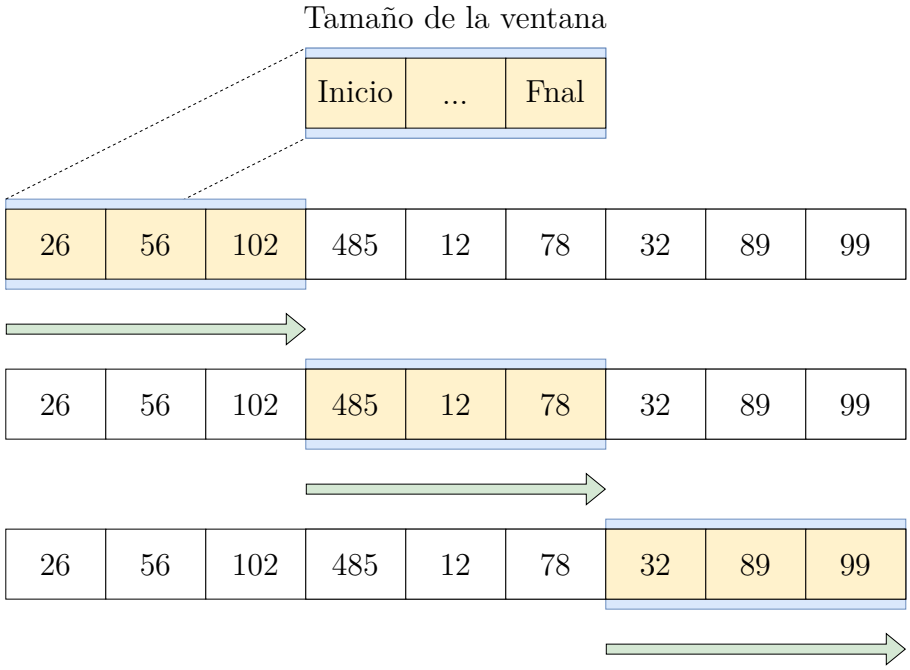
Consiste en tener dos punteros que indiquen el inicio y el final cada bloque con un tamaño fijo, esto es el tamaño de la ventana. Luego el arreglo es recorrido de forma iterativa con el objetivo de realizar alguna operación o modificación con los datos obtenidos en cada bloque. En la figura 7, se observa su funcionamiento.

Una situación bastante común la describe Sryheni (2020) cuando es necesario, por ejemplo, obtener la suma máxima de un conjunto de bloques de tamaño k en un arreglo. Utilizar ésta estrategia, reduce la complejidad del algoritmo hasta $O(n)$. Por supuesto, los escenarios de implementación pueden ser muy variados y la estructura de datos más compleja (de ser solo números a matrices, objetos u objetos de matrices), sin embargo, lo cierto es que es una técnica muy útil para ahorrar tiempo de ejecución cuando desarrollan algoritmos de esta naturaleza.

[8] A pesar que el énfasis de esta tesis está orientados a algoritmos metaheurísticos, es importante mostrar los pasos previos que se realizaron con el paso de los años hasta tener los métodos actuales.

Figura 7

Funcionamiento del algoritmo de ventana corrediza



Nota. El algoritmo de ventana corrediza se puede utilizar en múltiples contexto para resolver problemas iterativamente. Fuente: Elaboración propia.

3.2.2. Algoritmo de búsqueda anticipada

Es parte de los algoritmos de backtracking que permiten prever los efectos de elegir una variable u otra dentro de un espacio de soluciones. Esta técnica reduce en gran medida los tiempos de búsqueda para una solución parcial. Aunque haya sido pensado originalmente para problemas relacionados con grafos es posible utilizarlo en otros contextos como una técnica algorítmica válida.

Es importante entender que, en esencia, el backtracking es un enfoque sistemático que se basa en la recursión y la retrotracción para buscar soluciones de manera incremental, descartando aquellas ramas que no conducen a una solución válida.

Su procedimiento algorítmico usualmente se caracteriza por:

- **Exploración exhaustiva**: El algoritmo explora todas las posibles opciones en cada paso, retrocediendo cuando se alcanza un callejón sin salida.

- **Recursión**: Por lo general, se implementa de manera recursiva, lo que permite dividir el problema en subproblemas más pequeños y resolverlos de manera incremental.
- **Retroceso (backtrack)**: Cuando se llega a una solución inválida, el algoritmo retrocede (backtrack) a la decisión anterior y prueba una opción diferente.
- **Uso de una estructura de datos de pila (stack)**: Se utiliza una pila para mantener un registro de las decisiones tomadas en cada paso, lo que facilita el retroceso cuando sea necesario.

3.3. Algoritmos metaheurísticos

Etimológicamente, las palabras meta y heurístico vienen de los vocablos griegos: meta que significa *más allá* o *nivel superior* y heurístico definido como el arte de descubrir o plantear nuevas soluciones. Usualmente se cita al Dr. Fred Glover como el creador del término en 1986 con su trabajo Future paths for integer programming and links to artificial intelligence.

En principio, las metaheurísticas más que algoritmos son un conjunto de técnicas o metodologías que permiten aproximar soluciones de problemas de optimización, que por naturaleza son considerados complejos. Un factor clave que permite diferenciar estas técnicas de las heurísticas es su carácter estocástico.

Por definición, si existe un algoritmo A que resuelve un problema A se tiene un conjunto de entradas I que corresponden respectivamente a un conjunto de salidas O, es decir, para cada entrada i_k existe una salida o_k. Sin embargo, el carácter estocástico de un algoritmo metaheurístico permite afirmar que para una entrada i_k no necesariamente siempre existirá una misma salida o_k.

De acuerdo con Osman y Laporte (1996) citado en Pelta Mochcovsky (2013) el término metaheurística se define como “un proceso iterativo que guía una heurística subordinada, combinando de forma inteligente diferentes conceptos para explorar y explotar el espacio de búsqueda”. (p. 21).

Sin embargo, Du y Swamy (2016) lo define como:

Definición 15

Un procedimiento o heurística de nivel superior diseñado para encontrar, generar o seleccionar un procedimiento o heurística de nivel inferior (algoritmo de búsqueda parcial) que pueda proporcionar una solución suficientemente buena a un problema de optimización. Al buscar en un gran conjunto de soluciones factibles, la metaheurística puede encontrar a menudo buenas soluciones con menos esfuerzo computacional que los métodos basados en el cálculo, o la heurísticas simples (Du y Swamy, 2016, p. 10).

En su defecto, casi cualquier planteamiento sistemático conjugado con una componente de aleatoriedad podría considerarse un algoritmo metaheurístico. Es decir, pueden existir multitud de estrategias diferentes (unas más formales que otras) que permitan aproximar una solución la solución óptima para cualquier problema. Por esa razón, la literatura especializada ofrece un catálogo amplio de procedimientos que a menudo plantean nuevos puntos de vista sobre el problema a tratar.

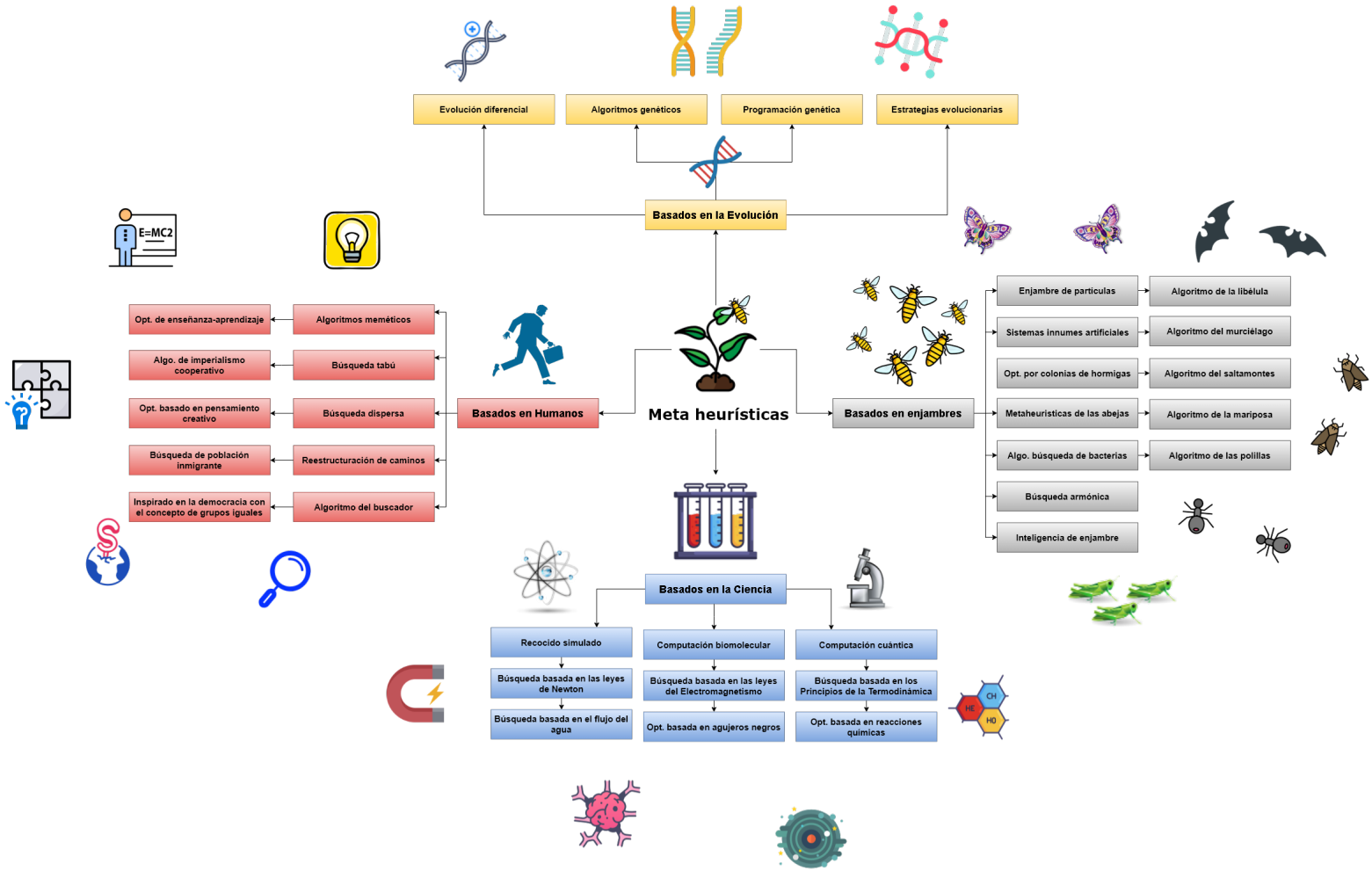
En este sentido es importante destacar que existe una gran cantidad de meta heurísticas diseñadas para tratar conjuntos o familias de problemas en concreto. Dentro de este enorme espectro de ideas hay un subconjunto interesante de meta heurísticas que siguen comportamientos inspirados en la naturaleza. Todo esto, por supuesto, bajo la idea de que la propia naturaleza tiene una tendencia intrínseca a mejorar continuamente y adaptarse.

Uno de los principales exponentes de este movimiento es Yang y He (2019) cuyas compilaciones y publicaciones han inspirado a la comunidad a continuar desarrollando estos conceptos que han resultado ser claves para la resolución de problemas reales de enorme complejidad. En la figura 8 se muestra una clasificación generalizada muy compacta de las meta heurísticas basadas en la naturaleza, en el comportamiento humano y distintos hechos científicos.

Para efectos prácticos, en este estudio solo se abordarán aquellos que gozan de una alta popularidad dentro de la comunidad, tales como: algoritmo genético, algoritmo memético y algoritmo de colonia de hormigas.

Figura 8

Clasificación de los algoritmos metaheurísticos



Nota. Clasificación de los algoritmos metaheurísticos según Du y Swamy (2016) y Okwu y Tartibu (2020)

3.3.1. Algoritmo genético

Según Kramer (2017), un algoritmo genético es considerado como:

Definición 16

Una búsqueda heurística aplicable a una amplia cantidad de problemas de optimización. Esta flexibilidad los hace atractivos para muchos problemas de optimización en la práctica. La evolución es la base de los algoritmos genéticos. (Krammer, 2017, p.11).

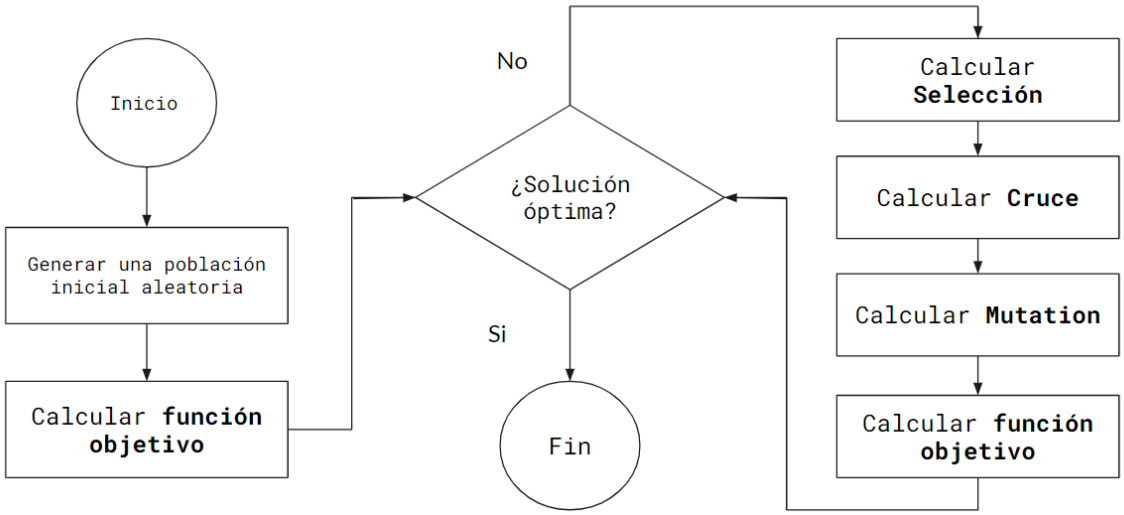
Los algoritmos genéticos son la expresión natural del comportamiento de los genes en la naturaleza y un reflejo de cómo afectan a una población de individuos de generación en generación. En la figura 9 se muestra la idea básica del funcionamiento de un algoritmo genético. En este caso, se trata de un planteamiento secuencial basado en una premisa de aleatoriedad en donde el propio de sistema trata de buscar una salida óptima de acuerdo a sus parámetros internos.

Este sistema está compuesto por padres e hijos y las distintas interacciones entre dichos actores. Por supuesto, al tratarse de una simulación de considerarse un procedimiento controlado y que cumpla unas condiciones mínimas para automatizarlo y aplicarlo a un problema real. De hecho, Hillier y Lieberman (2010) plantean cinco aspectos importantes que todo GA debe tener antes de su implementación:

- **Tamaño de la población:** Aunque este parámetro podría variar según las dimensiones del problema, es importante iniciar con valores pequeños para posteriormente incrementar su dimensión y observar los cambios en su comportamiento.
- **Selección de padres:** De la población inicial se selecciona a un subconjunto de individuos de forma aleatoria, estos serán considerados los más aptos. Lo ideal es formar parejas para posteriormente cruzarlos y así obtener nuevas salidas.
- **Transferencia de características (genes) de padres a hijos:** A este proceso también se le conoce como *cruce*. Es decir, es un pequeño algoritmo interno que normalmente lo define el investigador o el problema de estudio, en el cual se calculan los nuevos genes basados en diversos criterios.

Figura 9

Diagrama de flujo básico de un GA



Nota. Esta es una descripción gráfica estándar de un algoritmo genético para resolver un problema de optimización. Fuente: Elaboración propia.

- **Tasa de mutación:** Como no todo en la naturaleza es perfecto, también se debe considerar un factor de mutación en el modelo de tal manera que dicha tasa afecte a los nuevos genes en una proporción razonable. Por lo general este valor está entre 0 y 1 y en algunas ocasiones suele ser aleatorio.

- **Regla de detención:** Todo proceso iterativo debe tener un fin. Sin embargo, esta condición puede estar basada en algún parámetro de mejora que si al cabo de n iteraciones no logra superarse el proceso es detenido y se extraen los resultados obtenidos para su posterior análisis. (p. 591)

Quizá estas definiciones no sean matemáticamente rigurosas, sin embargo a pesar de ello se ha logrado demostrar que al menos en su forma base este tipo de procedimientos tienen una convergencia bien definida, Rudolph (1994). Aunque esto suele ser más fácil de entender en la práctica.

Un ejemplo clásico e ilustrativo sobre su implementación y codificación es adivinar una contraseña (como una secuencia de caracteres), utilizando las técnicas anteriormente descritas. El procedimiento se describe a continuación.

En el paso 1 se inicia con el conjuntos de genes a trabajar y el objetivo que se quiere obtener.

Figura 10

GA - Paso 1

```
geneSet = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!."
target = "Hello World!"
```

Nota. Configuración inicial del algoritmo genético.
Tomado de Sheppard (2017)

Posteriormente, en el paso 2 se establece un procedimiento para escoger de forma aleatoria a los padres, tomando como entrada el tamaño de la función objetivo y retornando una cadena con la secuencia de genes.

Figura 11

GA - Paso 2

```
import random

def generate_parent(length):
    genes = []
    while len(genes) < length:
        sampleSize = min(length - len(genes), len(geneSet))
        genes.extend(random.sample(geneSet, sampleSize))
    return ''.join(genes)
```

Nota. Función que genera de forma aleatoria a los padres.
Tomado de Sheppard (2017)

Luego se desarrolla una función que se encargará de encontrar el óptimo, esta es la función a optimizar. Recibe como entrada una cadena de texto y retorna un valor numérico el cual se utilizará en la regla de detención.

Figura 12

GA - Paso 3

```
def get_fitness(guess):
    return sum(1 for expected, actual in zip(target, guess)
               if expected == actual)
```

Nota. Función objetivo o de optimización.
Tomado de Sheppard (2017)

Después se define una función de mutación, en este caso recibe una cadena de entrada y también retorna otra cadena de texto. Su funcionalidad consiste en generar dos pequeños conjuntos aleatorios, posteriormente alternarlos y luego verificar si coinciden unos con otros.

Figura 13

GA - Paso 4

```
def mutate(parent):
    index = random.randrange(0, len(parent))
    childGenes = list(parent)
    newGene, alternate = random.sample(geneSet, 2)
    childGenes[index] = alternate \
        if newGene == childGenes[index] \
        else newGene
    return ''.join(childGenes)
```

Nota. Función de mutación.
Tomado de Sheppard (2017)

También es importante generar una función que permita visualizar los resultados de cada iteración para comparar sus cambios. En función display se muestra la cantidad de tiempo que ha tardado el programa hasta ese punto.

Luego en el programa principal se ha definido un ciclo infinito while hasta que la regla de detención definida anteriormente se cumpla en la sentencia break.

Figura 14

GA - Paso 5

```
import datetime
...
def display(guess):
    timeDiff = datetime.datetime.now() - startTime
    fitness = get_fitness(guess)
    print("{}\t{}\t{}".format(guess, fitness, timeDiff))
```

Nota. Función que muestra los resultados del algoritmo genético. Tomado de Sheppard (2017)

Figura 15

GA - Paso 6

```
while True:
    child = mutate(bestParent)
    childFitness = get_fitness(child)
    if bestFitness >= childFitness:
        continue
    display(child)
    if childFitness >= len(bestParent):
        break
    bestFitness = childFitness
    bestParent = child
```

Nota. Iteración que evalúa si se alcanzó el óptimo en la función objetivo con cada mutación generada. Tomado de Sheppard (2017)

Finalmente, se muestran los resultados (que por cierto, siempre serán diferentes) en el cual el programa parte de una cadena aleatoria y va reconstruyendo el texto progresivamente hasta llegar al objetivo inicial. De esta forma, los algoritmos genéticos permiten establecer un marco de trabajo para resolver un problema lo más rápido posible utilizando en su mayoría técnicas de carácter estocástico.

Figura 16

Resultados del GA

```
ftljCDPvhasn    1   0:00:00
ftljC Pvhasn    2   0:00:00
ftljC Pohasn    3   0:00:00.001000
HtljC Pohasn    4   0:00:00.002000
HtljC Wohasn    5   0:00:00.004000
Htljo Wohasn    6   0:00:00.005000
Htljo Wohas!    7   0:00:00.008000
Htljo Wohls!    8   0:00:00.010000
Heljo Wohls!    9   0:00:00.013000
Hello Wohls!    10  0:00:00.013000
Hello Wohld!    11  0:00:00.013000
Hello World!    12  0:00:00.015000
```

Nota. Resultados iterativos mostrados hasta alcanzar el óptimo en el paso 12. Tomado de Sheppard (2017)

3.3.2. Algoritmo memético

Según Moscato y Cotta (2003) los algoritmos meméticos son:

Definición 17

Son técnicas de optimización que combinan sinérgicamente conceptos tomados de otras metaheurísticas, tales como la búsqueda basada en poblaciones (como en los algoritmos evolutivos), y la mejora local (como en las técnicas de seguimiento del gradiente) (Moscato y Cotta, 2003, p. 1).

Cabe mencionar que la idea de estos algoritmos es imitar el comportamiento de algunos fenómenos físicos o naturales con el fin de orquestar un procedimiento de optimización que sea eficiente para la resolución de un determinado problema.

En la figura 17 se muestra la plantilla base de un algoritmo memético. Observe que su estructura es muy similar a un GA. De hecho, es bastante común que estos dos procedimientos trabajen en conjunto, especialmente cuando es posible plantear las posibles soluciones del problema en términos de una población.

Figura 17

Elementos básico de un algoritmo memético

```
FUNC MA (↓ tamañoPob : ℕ,
         ↓ ops : Operador[ ]) → Agente
Variables
    pob : Agente[ ];
Inicio
    pob ← Iniciar-Poblacion(tamañoPob);
    REPETIR
        pob ← Paso-Generacional (pob, ops)
        SI Convergencia(pob) ENTONCES
            pob ← Reiniciar-Poblacion(pob);
        FINSI
    HASTA QUE Terminacion-MA()
    DEVOLVER i-esimo-Mejor(pob, 1);
Fin
```

Nota. Un algoritmo memético tiene como objetivo imitar el comportamiento de ciertos fenómenos físicos o biológicos. Tomado de Moscato y Cotta (2003)

A pesar de su flexibilidad para trabajar en conjunto en la resolución de problemas complejos, ambos procedimientos tienen notables diferencias, tal como lo muestra Moscato y Cotta (2003) en su cuadro comparativo de la figura 18.

3.3.3. Algoritmo de colonia de hormigas

La optimización por colonias de hormigas por sus siglas en inglés (ACO) es una de las tantas familias de metaheurísticas inspiradas en la naturaleza de inteligencia de enjambres. En la figura 19 se muestran las distintas disciplinas asociadas a dicho modelo.

Figura 18

Comparativa entre GA y MA

GA	
codificación	esquemas, cadenas lineales, alfabetos predefinidos
individuo	solución al problema
cruce	intercambio no-guiado de información
mutación	introducción aleatoria de nueva información
MA	
representación	formas, no-linealidad, cercanía al problema
agente	solución/ones al problema + mecanismo mejora local
recombinación	intercambio guiado de información
mutación	introducción sensible de nueva información
mejora local	aprendizaje lamarckiano

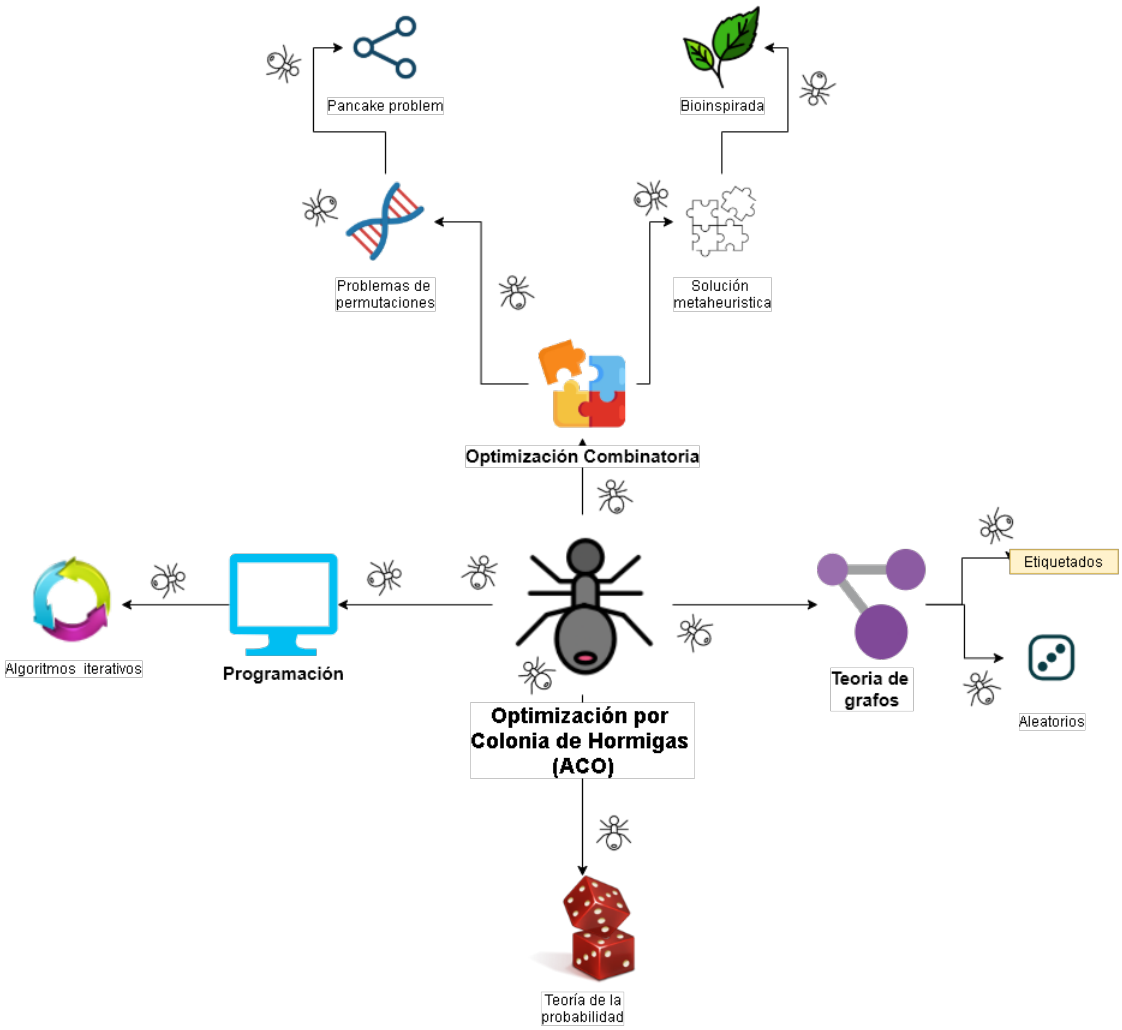
Nota. Principales características de los algoritmos genéticos y meméticos. Tomado de Moscato y Cotta (2003)

Se sabe que las hormigas antes de localizar una fuente de alimentos de calidad realizan un recorrido aleatorio en las cercanías de su nido. En algún momento alguna de ellas encontrará el alimento deseado y para dejar una referencia a sus compañeras segregan a su paso ciertas cantidades de feromonas [9]. A su vez, las demás hormigas también hacen lo mismo dejando, tras muchos viajes y rutas distintas, el camino más eficiente para que las compañeras en el nido también salgan y apoyen con la carga de la comida.

[9] De la raíz del gr. phérein 'llevar' y la t. de hormone 'hormona'. Sustancia química cuya liberación al medio es producida por un organismo. https://dle.rae.es/feromona?m=form

Figura 19

Disciplinas relacionadas con ACO



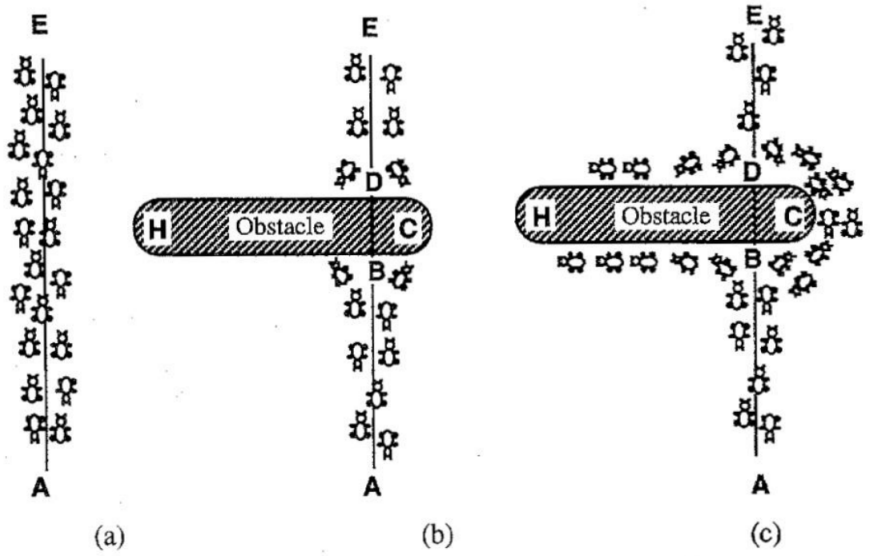
Nota. Principales ramas técnicas y científicas relacionadas con ACO. Fuente: Elaboración propia.

En términos algorítmicos, esto significa que cada iteración de feromonas segregadas por las hormigas construye poco a poco una ruta óptima entre dos puntos teniendo en cuenta que la inicialización es naturalmente estocástica [10]. En la figura 20 se puede ver un ejemplo de su comportamiento real.

[10] Teoría estadística de los procesos cuya evolución en el tiempo es aleatoria, tal como la secuencia de las tiradas de un dado. https://dle.rae.es/estocastico?m=form

Figura 20

Un ejemplo con hormigas reales



Nota. (a) Las hormigas siguen un camino entre los puntos A y E. (b) Se interpone un obstáculo; las hormigas pueden elegir rodearlo siguiendo uno de los dos caminos diferentes con igual probabilidad. (c) En el camino más corto se depositan más feromona. Tomado de Dorigo et al. (1996)

En este sentido, Dorigo y Blum (2005) definen a los algoritmos ACO como procedimientos de búsqueda estocástica. Su componente central es el modelo de las feromonas, que se utiliza para muestrear probabilísticamente el espacio de búsqueda.

Es importante resaltar que este método está basado en un grafo ponderado. En ese escenario, las hormigas se movilizan metafóricamente [11] sobre el grafo buscando un camino óptimo basado en los distintos parámetros que le brindan dinamismo al propio modelo.

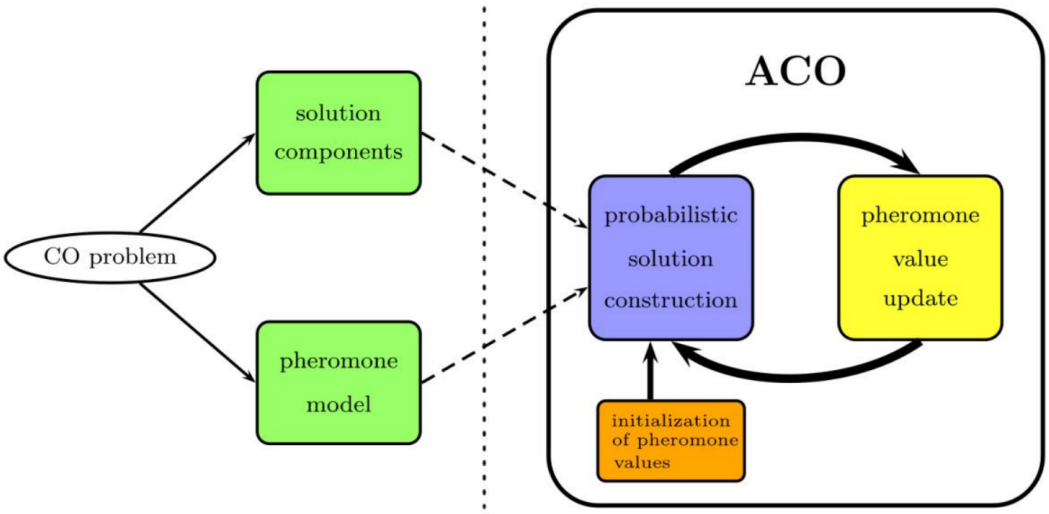
Usualmente, la terminología y notación utilizada por los autores suele ser confusa. Sin embargo, en este apartado se brindará un resumen sencillo para entender el modelo matemático que hay detrás de esta metaheurística.

[11] Para una visualización más clara y completa del modelo visite el enlace en Github. https://poolik.github.io/visual-aco/#/visualisation

Blum (2005) actualizó no solo el marco de trabajo (ver figura 21) de dicho modelo sino algunas de las ecuaciones inicialmente propuestas por Dorigo et al. (1996).

Figura 21

Marco de trabajo de ACO



Nota. Esta es una descripción de un marco de trabajo clásico para el desarrollo de algoritmos de optimización basados en ACO. Tomado de Blum (2005)

Por simplicidad, se han tomado como referencia el marco de trabajo de Blum (2005) y se ha modificado la notación.

El modelo de feromonas está compuesto por los siguientes elementos:

Definición 18

Probabilidad de que una hormiga seleccione un camino:

$$P_{xy} = \frac{\tau(C_i)\eta(C_i)}{\sum \tau(C_i)\eta(C_i)}$$

(p. 360)

Definición 19

Cantidad de feromonas en cada iteración:

$$\tau_{xy} = (1 - \rho)\tau_{xy} + \sum \Delta\tau_{xy}$$

(p. 361)

Definición 20

Aporte de feromonas por cada hormiga

$$\Delta\tau_{xy} = \begin{cases} \frac{Q}{L_k} & \text{si se usa xy} \\ 0 & \text{si no se usa} \end{cases}$$

(p. 361)

En donde:

- Término de visibilidad: $\eta(C_i) = \frac{1}{d(Ci)}$
- Número de feromonas: $\tau(C_i)$
- Parámetro de aprendizaje: Q
- Costo camino-hormiga: L_k
- Ratio de evaporación: ρ
- i-ésimo nodo del grafo: C_i

Antes de continuar es importante aclarar que durante la ejecución de la metaheurística se tendrá en cuenta un porcentaje o ratio de evaporación de las feromonas. Este hecho tiene un impacto significativo sobre el modelo, ya que implica que el tiempo para mantener ese camino de feromonas activo es limitado si es el incorrecto, pero se mantendrá más en el tiempo si dicho camino está cercano al óptimo. Como lo indica la figura estos valores de evaporación y de feromonas se actualizarán en cada iteración basados en el registro probabilístico previo.

Por otro lado, la cantidad de algoritmos e implementaciones sobre esto modelo es muy amplia ya que cada autor lo modifica de acuerdo a las necesidades de su problema de estudio.

3.4. Ordenes de complejidad computacional

Usualmente cuando se desarrolla una solución algorítmica para un problema, es muy común cuestionar el grado de sencillez, claridad y optimización del procedimiento en sí mismo.

Existen muchas maneras informales que permiten realizar un análisis de la calidad de una solución desarrollada, sin embargo, el método más común es el análisis asintótico, representado en una función continua que normalmente calcula la cantidad del tiempo computacional invertido en dependencia del tamaño de las entradas del algoritmo. Esta idea fue planteada inicialmente por el Dr. Donald Knuth en los años 60s y desde entonces se ha convertido en una de las bases para el análisis de algoritmos moderno.

Aunque, en términos simples, la complejidad computacional trata sobre cómo un algoritmo se acopla a una función para así determinar su rendimiento, ya sea en términos de tiempo o memoria, sin antes siquiera haberlo programado. La figura 22 muestra las funciones más conocidas para este tipo de análisis.

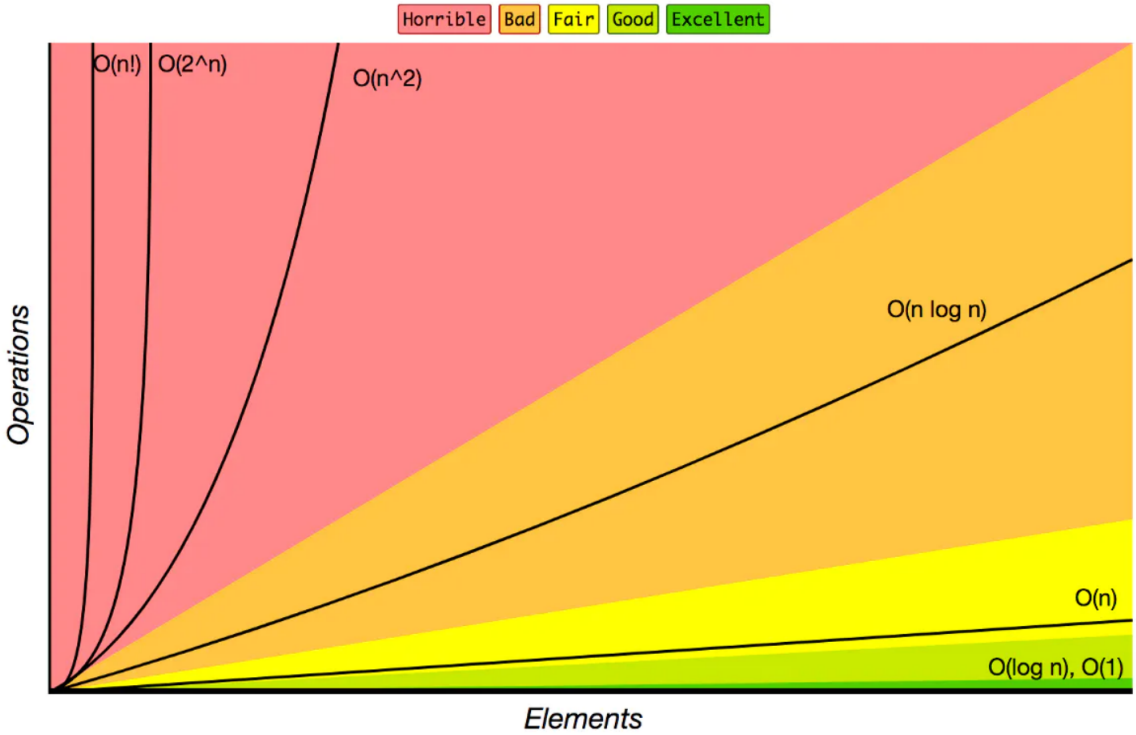
Por otro lado, existen al menos tres enfoques distintos que permiten obtener una comprensión clara y cuantificable antes de su codificación.

3.4.1. Análisis de casos

El peor caso ocurre cuando un algoritmo cumple exhaustivamente todos los pasos sin excepción. En términos de memoria y tiempo este escenario sería el indeseable y el que se debe de evitar siempre que sea posible.

Figura 22

Funciones más comunes en el análisis de algoritmos



Nota. Funciones base para análisis asintótico en teoría de la complejidad computacional. Tomado de Haines (2017)

Según Baase et al. (2002), la complejidad el peor caso se define como:

Definición 21

Complejidad del peor caso:

$$W(n) = max\{t(I)|I \in D_n\}$$

(Baase et al., 2002, p. 34)

Donde, D_n es el numero del entradas para el problema, I es un elemento de D_n y $t(I)$ el número de operaciones básicas que ejecuta el algoritmo con la entrada I. $W(n)$ es el número máximo de operaciones que ejecuta el algoritmo con cualquier entrada de tamaño n.

La definición 21 implica manejar escenarios en donde, por ejemplo, en donde ciclo for sea recorrido exhaustivamente desde 1 hasta n, sin tener una condición de parada válida. Este es el caso del algoritmo para encontrar el número mayor en una arreglo unidimensional. Si se realiza un recorrido de todo el arreglo y resulta que el número mayor está al final, ahí se podría afirmar que el peor caso para dicho algoritmo es $O(n)$, es decir, su comportamiento es lineal pero a la vez no ha sido tan rápido como normalmente se espera.

Definición 22

Complejidad del caso promedio:

$$A(n) = \sum_{I \in D_n} Pr(I)t(I)$$

(Baase et al., 2002,p. 35)

Por otro lado, el análisis del caso promedio, o lo que en teoría podría ocurrir basado en la probabilidad de $Pr(I)$ de que una entrada sea o no tomada en cuenta para el algoritmo. Sin embargo, este valor no es posible calcularlo analíticamente, es decir, no existen fórmulas o ecuaciones que definan un procedimiento claro para obtener $Pr(I)$, puesto que suele ser un valor arbitrario puesto por el investigador basado en su experiencia.

Aunque en términos prácticos, realizar un análisis del peor caso es más conveniente ya que permite validar las principales debilidades del algoritmo con el objetivo de reducir cantidad de fallos y casos especiales al mínimo.

3.4.2. Notación asintótica

La notación asintótica es una representación matemática del comportamiento de un algoritmo. En la figura 23 muestra a los tres grandes conjuntos de funciones según su tasa de crecimiento.

A continuación se describen las definiciones formales para dichos conjuntos, con base en teoría de funciones reales con variable real:

Definición 23

Conjunto $O(g)$: Sea g una función de los enteros no negativos a los números reales positivos. Entonces, $O(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que para alguna constante real $c > 0$ y alguna constante entera no negativa n_0, $f(n) \leq cg(n)$ para toda $n \geq n_0$. (Baase et al., 2002, p. 45)

Definición 24

Conjunto $\Omega(g)$: Sea g una función de los enteros no negativos a los números reales positivos. Entonces, $\Omega(g)$ es el conjunto de funciones f, también de los enteros no negativos a los números reales positivos, tal que para alguna constante real $c > 0$ y alguna constante entera no negativa n_0, $f(n) \geq cg(n)$ para toda $n \geq n_0$. (Baase et al., 2002, p. 47)

Definición 25

Conjunto $\Theta(g)$: Sea g una función de los enteros no negativos a los números reales positivos. Entonces, $\Theta(g) = O(g) \cap \Omega(g)$, es decir, el conjunto de funciones que están tanto en $O(g)$ como en $\Omega(g)$. (Baase et al., 2002, p. 48)

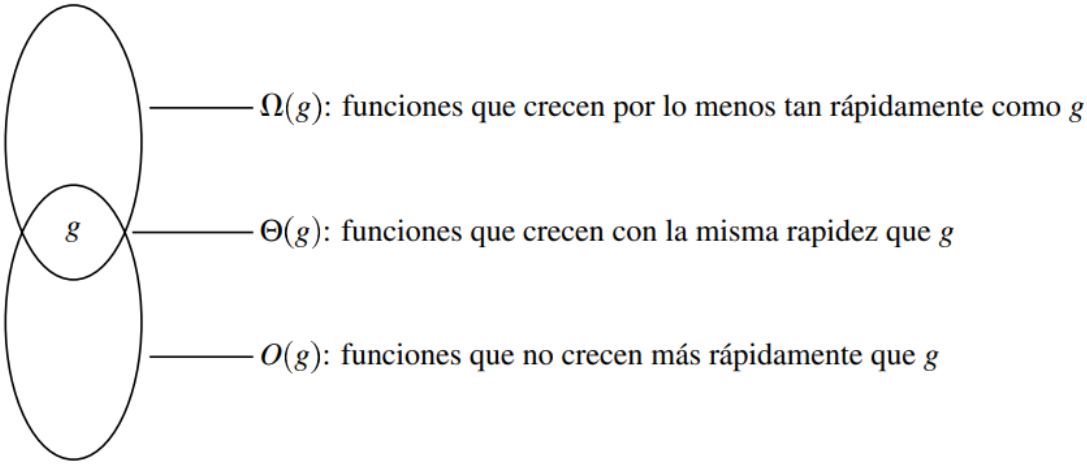
En la figura 24 se muestra un ejemplo de 5 algoritmos con diferentes órdenes de complejidad, cuyo impacto en el tiempo de ejecución está determinado con la función asintótica con la que se encuentra relacionado.

3.5. MIN-SBPR es NP-Hard

Como se describió en secciones anteriores, este problema cuenta con un precedente muy importante y es que se demostró formalmente a qué clase de complejidad pertenecía. Este hecho permitió desarrollar nuevas líneas de investigación que plantearan soluciones no necesariamente por la vía de la exactitud y la rigurosidad matemática.

Figura 23

Orden de complejidad: Omega grande (Ω), theta grande (Θ) y o grande (O).



Nota. Clasificación de crecimiento asintótico según la notación Big O. Tomado de Baase et al. (2002)

Figura 24

Ejemplos del crecimiento asintótico de funciones comunes

Algoritmo	1	2	3	4	
Función de tiempo (μs)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	2^n
Tamaño de la entrada (n)			*Tiempo para resolver*		
10	0.00033 s	0.0015 s	0.0013 s	0.0034 s	0.001 s
100	0.003 s	0.03 s	0.13 s	3.4 s	$4 \cdot 10^{16}$ años
1,000	0.033 s	0.45 s	13 s	0.94 h	
10,000	0.33 s	6.1 s	22 min	39 días	
100,000	3.3 s	1.3 min	1.5 días	108 años	
Tiempo permitido			*Tamaño de entrada máximo soluble (aprox.)*		
1 segundo	30,000	2,000	280	67	20
1 minuto	1,800,000	82,000	2,200	260	26

Nota. Tiempo computacional que toman ciertas funciones en dependencia de su crecimiento asintótico. Tomado de Baase et al. (2002)

Sin embargo, en esta sección se mostrarán los elementos más importantes que el Dr. Laurent Bulteau tomó en consideración para realizar su demostración en 2014. Antes, es importante destacar que esta demostración aplica para el caso de minimización del problema y su variante sin signo. El objetivo de la demostración está basado en mostrar que la cantidad de puntos de interrupción en una permutación es una cota inferior en relación a la cantidad de vueltas necesarias que hay que aplicar para ordenarla, decidir si ese límite está lo suficientemente ajustado es considera NP-hard[12].

3.5.1. Conceptos y notación

Las matemáticas son bellas porque se construyen a sí mismas. Eso lo demostraron estos autores con su notable trabajo, quienes establecieron las bases para la notación estándar que actualmente sigue en uso dentro de la literatura especializada.

Según Bulteau et al. (2012), la siguiente lista corresponde a todos los nuevos conceptos y herramientas matemáticas desarrolladas en la demostración con el objetivo de plantear una propuesta robusta y consistente matemáticamente.

- Vueltas (flip)
- Bloqueo (deadlock)
- Eficiente (efficient)
- Eficientemente ordenable (efficiently sortable)
- Ruta (path)
- Secuencias gadget (gadget sequences)
- Gadget de muelle (dock gadget)
- Gadget de cláusula (clause gadget)
- Gadget de bloqueo (lock gadget)
- Gadget de gancho (hook gadget)

[12]Es un conjunto de problemas de decisión que como mínimo son tan difíciles como un problema NP. https://es.wikipedia.org/wiki/NP-hard

- Gadget de orquilla (fork gadget)
- Literales (literals)
- Variables (variables)
- Cláusulas (clauses)
- Reducciones (reductions)

Todas estas, a excepción de las últimas cuatro, son formaciones especiales o patrones encontrados en las permutaciones que cumplen con las condiciones del problema. El caso más sencillo es el concepto de vuelta o *flip*, el cual hace referencia a una permutación que pueda dividirse en dos partes, girando la primera parte de derecha a izquierda hasta invertir por completo el orden de sus elementos.

Los últimos cuatro conceptos son realmente curiosos, ya que, con todos los elementos previamente definidos, los autores realizaron un planteamiento muy similar a un lenguaje de programación basado en su formulación matemática, utilizando variables, literales, cláusulas y reducciones (operaciones) se llegó a la conclusión de que el problema en términos combinatorios es inmanejable en términos computacionales para entradas muy grandes.

Por otro lado, utilizaron una notación vectorial para representar secuencias o permutaciones y todas sus propiedades asociadas (ver figura 25). También, utilizaron la notación de conjuntos para formular la mayoría de sus definiciones.

Todos estos conceptos se reducen a un problema más concreto como 3-SAT, que se sabe que es NP-Hard. Esta reducción muestra que si existiera un algoritmo de tiempo polinómico para resolver el problema SBPR, entonces también existiría un algoritmo de tiempo polinómico para resolver el problema 3-SAT.

En general, el trabajo de Bulteau es valioso porque proporciona una prueba matemática rigurosa de la dureza computacional del problema de ordenación de permutaciones por inversión de prefijos y sus variaciones, que tiene implicaciones para el diseño y análisis de algoritmos en optimización combinatoria y biología computacional.

Figura 25

Compilación de todas las propiedades gadget

Dock gadget

$$\langle {}^{\star}\mathcal{I}_q^{p+1},\ X,\ \boldsymbol{D},\ Y\rangle \underset{4.}{\Longrightarrow} \langle X,\ \mathcal{I}_{q+2}^{p-1},\ Y\rangle$$

Lock gadget

$$\langle \boldsymbol{key},\ X,\ \boldsymbol{L},\ Y\rangle \underset{5.a}{\Longrightarrow} \langle X,\ L^{o},\ Y\rangle$$

$$\langle \boldsymbol{test},\ X,\ \boldsymbol{L^{o}},\ Y\rangle \underset{5.b}{\Longrightarrow} \langle X,\ \mathcal{I}_{p+12}^{p+1},\ Y\rangle$$

$$\langle \boldsymbol{test},\ X,\ \boldsymbol{L},\ Y\rangle \underset{5.c}{\rightarrow} \perp$$

Hook gadget

$$\langle \boldsymbol{take},\ X,\ \boldsymbol{G},\ Y,\ \boldsymbol{H},\ Z\rangle \underset{6.a}{\Longrightarrow} \langle Y,\ G',\ {}^{\star}X,\ H',\ Z\rangle$$

$$\langle \boldsymbol{put},\ X,\ \boldsymbol{G'},\ {}^{\star}Y,\ \boldsymbol{H'},\ Z\rangle \underset{6.b}{\Longrightarrow} \langle Y,\ G'',\ X,\ H'',\ Z\rangle$$

$$\langle \boldsymbol{G''},\ X,\ \boldsymbol{H''},\ Y\rangle \underset{6.c}{\Longrightarrow} \langle X,\ {}^{\star}\mathcal{I}_{p+12}^{p+1},\ Y\rangle$$

Fork gadget

$$\langle \boldsymbol{E},\ X,\ \boldsymbol{F},\ Y\rangle \underset{7.a}{\Longrightarrow} \left\{ \begin{array}{l} \langle X,\ F^{1},\ Y\rangle \\ \langle {}^{\star}X,\ F^{2},\ Y\rangle \end{array} \right\}$$

$$\langle \boldsymbol{F^{1}},\ Y\rangle \underset{7.b}{\Longrightarrow} \langle {}^{\star}\mathcal{I}_{p+15}^{p+1},\ Y\rangle$$

$$\langle \boldsymbol{F^{2}},\ Y\rangle \underset{7.c}{\Longrightarrow} \langle {}^{\star}\mathcal{I}_{p+15}^{p+1},\ Y\rangle$$

Literals gadget

$$\forall i \notin O \cup I,\ \langle \boldsymbol{key_i},\ X,\ \boldsymbol{\Lambda_I^O}\rangle \underset{8.a}{\Longrightarrow} \langle X,\ \Lambda_I^{O\cup\{i\}}\rangle$$

$$\forall i \in O,\ \langle \boldsymbol{test_i},\ X,\ \boldsymbol{\Lambda_I^O}\rangle \underset{8.b}{\Longrightarrow} \langle X,\ \Lambda_{I\cup\{i\}}^{O-\{i\}}\rangle$$

$$\forall i \notin O,\ \langle \boldsymbol{test_i},\ X,\ \boldsymbol{\Lambda_I^O}\rangle \underset{8.c}{\rightarrow} \perp$$

Variable gadget

$$\langle \boldsymbol{v},\ X,\ \boldsymbol{V},\ Y,\ \boldsymbol{\Lambda_I^O}\rangle \underset{9.a}{\Longrightarrow} \left\{ \begin{array}{l} \langle X,\ V^{1},\ Y,\ \Lambda_I^{O\cup P}\rangle \\ \langle X,\ V^{2},\ Y,\ \Lambda_I^{O\cup N}\rangle \end{array} \right\}$$

$$\langle \boldsymbol{V^{1}},\ X,\ \boldsymbol{D},\ Y,\ \boldsymbol{\Lambda_I^O}\rangle \underset{9.b}{\Longrightarrow} \langle X,\ \mathcal{I}_{p+31}^{p+1},\ Y,\ \Lambda_I^{O\cup N}\rangle$$

$$\langle \boldsymbol{V^{2}},\ X,\ \boldsymbol{D},\ Y,\ \boldsymbol{\Lambda_I^O}\rangle \underset{9.c}{\Longrightarrow} \langle X,\ \mathcal{I}_{p+31}^{p+1},\ Y,\ \Lambda_I^{O\cup P}\rangle$$

Clause gadget

$$\langle \boldsymbol{\gamma},\ X,\ \boldsymbol{\Gamma},\ Y,\ \boldsymbol{\Lambda_I^O}\rangle \underset{10.}{\Longrightarrow} \left\{ \begin{array}{l} \langle X,\ \Gamma^{1},\ Y,\ \Lambda_{I\cup\{a\}}^{O-\{a\}}\rangle \text{ iff } a \in O \\ \langle X,\ \Gamma^{2},\ Y,\ \Lambda_{I\cup\{b\}}^{O-\{b\}}\rangle \text{ iff } b \in O \\ \langle X,\ \Gamma^{3},\ Y,\ \Lambda_{I\cup\{c\}}^{O-\{c\}}\rangle \text{ iff } c \in O \end{array} \right\}$$

$$\langle \boldsymbol{\Gamma^{1}},\ Y,\ \Delta,\ Z,\ \boldsymbol{\Lambda_I^O}\rangle \underset{11.a}{\Longrightarrow} \langle Y,\ \mathcal{I}_{p+62}^{p+1},\ Z,\ \Lambda_{I\cup\{b,c\}}^{O-\{b,c\}}\rangle$$

$$\langle \boldsymbol{\Gamma^{2}},\ Y,\ \Delta,\ Z,\ \boldsymbol{\Lambda_I^O}\rangle \underset{11.b}{\Longrightarrow} \langle Y,\ \mathcal{I}_{p+62}^{p+1},\ Z,\ \Lambda_{I\cup\{a,c\}}^{O-\{a,c\}}\rangle$$

$$\langle \boldsymbol{\Gamma^{3}},\ Y,\ \Delta,\ Z,\ \boldsymbol{\Lambda_I^O}\rangle \underset{11.c}{\Longrightarrow} \langle Y,\ \mathcal{I}_{p+62}^{p+1},\ Z,\ \Lambda_{I\cup\{a,b\}}^{O-\{a,b\}}\rangle$$

Nota. Propiedades vectoriales y conceptos base necesarios para la demostración del Dr. Bulteau. Tomado de Bulteau et al. (2012)

Algoritmos heurísticos y metaheurísticos del problema SBPR

Capítulo 4

4. ALGORITMOS HEURÍSTICOS Y METAHEURÍSTICOS DEL PROBLEMA SBPR

En esta sección se presenta una breve descripción del pseudocódigo asociado a cada uno de los algoritmos seleccionados para su posterior análisis y comparación. Es importante señalar que cada algoritmo cuenta con un código de identificación único, además de su nombre correspondiente.

4.1. Heurísticas

4.1.1. Algoritmo de dias2014

El algoritmo 1 U. Dias et al. (2014) también conocido como "ventana deslizante" o "ventana corrediza" en inglés, es un método utilizado para identificar dinámicamente el último elemento de un arreglo a partir de una secuencia de soluciones dadas. La ventana deslizante es una secuencia de una secuencia-solución, y el algoritmo trata de encontrar la mejor manera de transformar los elementos dentro de esta ventana.

Este procedimiento está dividido en cinco partes:

1. El algoritmo comienza fijando el final de la ventana un elemento por delante del inicio de la ventana.

2. A continuación, itera a través de la secuencia de soluciones, buscando el mayor índice para el final de la ventana tal que la reducción de los elementos dentro de la ventana resulte en una permutación cuyo tamaño sea menor o igual a 12.

3. El algoritmo se detiene cuando encuentra el final de la ventana o cuando no hay más permutaciones en la secuencia solución que satisfagan la restricción de tamaño.

4. Durante este proceso, se utiliza una función llamada "reduce" para realizar una reducción o reducción parcial de los elementos dentro de la ventana, dependiendo del problema específico de reordenación del genoma que se esté abordando.

5. El algoritmo mantiene un registro de los elementos descartados de cada franja dentro de la ventana, lo que ayuda a recrear la permutación original.

La base de datos mencionada en este estudio es un componente clave en la construcción de la heurística general. Esta estructura almacena las soluciones exactas para todas las permutaciones posibles cuyos tamaños oscilan entre 1 y 12 unidades. Esta base de datos se utiliza para recuperar soluciones óptimas para instancias pequeñas del problema que posteriormente se utilizan para mejorar las soluciones proporcionadas por algoritmos no óptimos.

En general, para desarrollar una base de datos[13] o diccionario clave valor de este tipo, se deben seguir los siguientes pasos:

1. **Definir el ámbito de aplicación:** Determine los problemas específicos de ordenamiento del genoma para los que desea crear la base de datos. Esto podría incluir la ordenación por transposición, la ordenación por inversiones u otros problemas relacionados.

2. **Generar permutaciones:** Generar todas las permutaciones posibles de un tamaño dado. Por ejemplo, si está considerando permutaciones de tamaño hasta 12, necesitaría generar todas las permutaciones de longitudes 1 a 12.

3. **Calcular soluciones óptimas:** Para cada permutación generada, calcule la solución óptima para el problema específico de reordenación del genoma que está abordando. Esto puede implicar la implementación de algoritmos óptimos conocidos o el uso de soluciones existentes en la literatura.

4. **Almacenar soluciones:** Almacene las soluciones óptimas computadas en un formato estructurado que permita una recuperación eficiente basada en la permutación de entrada.

5. **Implementar un mecanismo de recuperación:** Desarrollar un mecanismo para recuperar las soluciones óptimas de la base de datos en función de la permutación de entrada. Esto podría implicar el uso indexación directa u otras técnicas para localizar rápidamente las soluciones almacenadas.

6. **Probar y validar:** Pruebe la base de datos consultándola con varias permutaciones de entrada y comparando las soluciones recuperadas con las soluciones óptimas conocidas para garantizar la precisión.

[13] Es importante aclarar que no se trata de una base de datos relacional como se puede entender tradicionalmente, en realidad se utiliza este término para referirse a un diccionario de datos u objeto dentro del programa.

De acuerdo con U. Dias et al. (2014) la complejidad computacional del primer algoritmo 1 es $O(n)$, donde n es el número de elementos en cada permutación de la secuencia de ordenamiento. Por otro lado, la complejidad del algoritmo 2 es $O(nm)$ debido a que el número de iteraciones realizadas en el ciclo *while* está limitado por el número de elementos de S y los pasos del ciclo que más tiempo consumen son $O(n)$.

```
1   # Algoritmo 1: Ventana corrediza - dias2014
2
3   datos: S = <S_0, S_1,...,S_m>, inicio, tamanio_base_datos
4   pi = inicio + 1
5
6   while True hacer:
7       pi = S_final * S_inicio
8       pi_reducido, descartado = reducir(pi)
9       if |pi_reducido| > tamanio_base_datos entonces:
10          return final - 1
11      end
12      si final = m entonces:
13          return final
14      end
15      final = final + 1
16  end
```

Algoritmo 1: Ventana corrediza - dias2014 - parte 1

```
# Algoritmo 2: Heuristica general - dias2014

datos: S = <S_0, S_1,...,S_m>, base_de_datos
inicio = 0

while True hacer:
    final = ventana_corrediza(S, inicio, base_de_datos.length)
    w = <S_inicio,...,S_final>
    pi = S_final * S_inicio
    pi_reducido, descartado = reducir(pi)
    base_de_datos_secuencia = base_de_datos.obtener(pi_reducido)
    if |base_de_datos_secuencia| < final - inicio + 1 entonces:
        w_1 = []
        foreach permutacion in base_de_datos_secuencia hacer:
            Q = expandir(permutacion, descartado)
            w_1.agregar(s_final*expandido)
        end
        S.reemplazar(w, w_1)
    end
    if final = m entonces:
        return S
    end
    inicio = inicio + 1
end
```

Algoritmo 2: Ventana corrediza - dias2014 - parte 2

4.1.2. Algoritmo de brito2019

En el algoritmo 3, también conocido como búsqueda anticipada o backtracking (en inglés) fue desarrollado por Brito et al. (2019) , y es utilizado para aproximar soluciones en problemas de ordenamiento del genoma.

A continuación también se explica el procedimiento del algoritmo 3 y su relación con el algoritmo 1:

1. **Funcionamiento del algoritmo de búsqueda anticipada:**

   - El algoritmo comienza con una permutación inicial y explora exhaustivamente el vecindario de la permutación, considerando todas las posibles reversiones y transposiciones.
   - En cada paso, se evalúan las operaciones de reversión y transposición en el vecindario para determinar cuál de ellas minimiza la distancia estimada entre la permutación actual y la permutación objetivo.
   - Para estimar la distancia, se utiliza un algoritmo en particular que calcula la cantidad de operaciones necesarias para ordenar la permutación actual en la permutación objetivo.
   - El algoritmo selecciona la operación que minimiza la distancia estimada y aplica dicha operación a la permutación actual, avanzando así hacia una solución más cercana a la óptima.

2. **Utilización para aproximar soluciones:**

   - Este algoritmo es utilizado para aproximar soluciones en problemas de reordenamiento genómico al explorar de manera exhaustiva en el espacio de soluciones y seleccionar las operaciones que minimizan la distancia estimada.
   - Al considerar todas las posibles reversiones en el vecindario de la permutación actual, el algoritmo busca mejorar la secuencia de ordenación de la permutación de manera iterativa.
   - A medida que avanza en la exploración del vecindario, el algoritmo busca encontrar una secuencia de operaciones que reduzca la distancia entre la permutación actual y la permutación objetivo, aproximando así una solución óptima o cercana al óptimo.

Sin embargo, el correcto funcionamiento de este procedimiento depende de varios factores tales como:

1. **Tamaño de la permutación:** El algoritmo puede ser más efectivo en permutaciones de tamaño moderado a grande, donde la exploración

exhaustiva del vecindario puede conducir a mejoras significativas en la secuencia de ordenación.

2. **Calidad del estimador de distancia:** Su rendimiento depende de la precisión con la que se estime la distancia utilizada para evaluar las operaciones de reversión en el vecindario de la permutación actual.

3. **Recursos computacionales disponibles:** Dado que el algoritmo implica una exploración exhaustiva y evaluación de múltiples operaciones en el vecindario de la permutación, su uso puede requerir recursos computacionales significativos, especialmente para permutaciones grandes.

En cuanto a la complejidad computacional, esta depende del procedimiento utilizado para estimar la distancia de las permutaciones en el vecindario. En general, esto se puede expresar como:

- **En la versión clásica:** $O(n^3 * alg(n))$, donde n es el tamaño de la permutación y $alg(n)$ es la complejidad del algoritmo de estimación de distancias.

- **En las versiones restringidas:** $O(n^2 * alg(n))$, donde n es el tamaño de la permutación y $alg(n)$ es la complejidad del algoritmo de estimación de distancias.

```
# Algoritmo 3: Busqueda anticipada - brito2019

datos: pi, algoritmo_distancia_mas_corta
resultado: secuencia_de_ordenamiento

permutacion_actual = pi
secuencia_de_ordenamiento = []

while permutacion_actual != ordenado hacer:
    vecindario = generar_vecindario(permutacion_actual)
    permutacion = algoritmo_distancia_mas_corta(vecindario)
    secuencia_de_ordenamiento.agregar(evento) *
operacion_de_reversion
    permutacion_actual = permutacion
end

return secuencia_de_ordenamiento
```

Algoritmo 3: Busqueda anticipada - brito2019

4.2. Metaheurísticas

Así como los procedimientos heurísticos tienen como objetivo eliminar lo más rápido posible la cantidad de puntos de interrupción en una permutación de tamaño n,las estrategias metaheurísticas comparten el mismo objetivo aunque estos procedimientos son un poco más sofisticados ya que en su mayoría se apoyan en técnicas estocásticas.

4.2.1. Algoritmo de soncco2013

En el algoritmo 4 Soncco-Álvarez y Ayala-Rincón (2013) adaptó el flujo de trabajo de un GA convencional al problema de estudio. Sin embargo lo novedoso de su planteamiento fue utilizar el concepto de *grafos con puntos de interrupción*. A pesar de ello, el algoritmo genético final después de haberse ejecutado n generaciones posee una complejidad de $O(n^3 logn)$.

Sin embargo, este algoritmo fue diseñado para la ordenación de permutaciones con signo, el cual es un caso recíproco del problema que se aborda en esta tesis.

Las permutaciones sin signo son secuencias de enteros positivos distintos, y la ordenación por inversiones para permutaciones sin signo consiste en encontrar la secuencia más corta de inversiones para transformar una permutación en otra.

Por otro lado, las permutaciones con signo pueden tener enteros positivos o negativos, y la ordenación por inversiones para permutaciones con signo consiste en encontrar la secuencia más corta de inversiones para transformar una permutación con signo en otra, teniendo en cuenta los signos de los enteros.

La diferencia radica en la complejidad adicional que introducen los signos en las permutaciones con signo, que afecta a la forma en que deben aplicarse las inversiones para transformar una permutación en otra. Esta complejidad adicional hace que la ordenación por inversiones para permutaciones con signo sea un problema diferente de la ordenación por inversiones para permutaciones sin signo.

A continuación se explica paso a paso el funcionamiento del algoritmo:

1. **Inicializar la población**: El algoritmo comienza generando una población inicial de soluciones candidatas. Cada solución candidata es una permutación de la secuencia de entrada, y la población se genera aleatoriamente.

2. **Evaluar la aptitud de cada solución candidata**: El algoritmo evalúa cada solución candidata utilizando la métrica de la distancia de inversión. La distancia de inversión es el número mínimo de inversiones de prefijo necesarias para transformar una permutación en otra.

3. **Selección de progenitores para la reproducción**: El algoritmo selecciona a los padres para la reproducción en función de su aptitud. Los individuos más aptos tienen una mayor probabilidad de ser seleccionados como padres.

4. **Reproducir**: El algoritmo reproduce los padres seleccionados para generar una nueva población de soluciones candidatas. El proceso de reproducción consiste en aplicar los operadores de cruce y mutación a los padres seleccionados.

5. **Evaluación de la aptitud de la nueva población**: El algoritmo evalúa la aptitud de la nueva población de soluciones candidatas utilizando la métrica de distancia de inversión.

6. **Seleccionar la mejor solución**: El algoritmo selecciona la mejor solución de la población actual de soluciones candidatas. Esta solución representa la permutación que requiere el mínimo número de inversiones de prefijos para transformar la secuencia de entrada en el orden deseado.

7. **Repetir**: El algoritmo repite los pasos 3 a 6 hasta que se cumple un criterio de parada. El criterio de parada puede ser un número máximo de generaciones, un umbral mínimo de aptitud o una combinación de ambos.

Es importante recordar que este procedimiento utiliza la métrica de la distancia de inversión para evaluar la aptitud de cada solución candidata y selecciona los individuos más aptos para su reproducción. El algoritmo repite este proceso hasta que se cumple un criterio de parada, y la mejor solución encontrada representa la permutación que requiere el mínimo número de inversiones de prefijos para transformar la secuencia de entrada en el orden deseado.

```
1 # Algoritmo 4: Algoritmo genetico con eliminacion de puntos de
    interrupcion - soncco2013
2
3 datos: pi, mejorar # mejorar es de tipo booleano
4 retultado: numero_de_reversiones_para_ordenar_pi
5 if mejorar entonces:
6     algirtmo_reviersion(pi)
7 end
8 generar_poblacion_inicial(pi)
9
10 # Evaluar funcion objetivo para cada individuo de la poblacion
11 for i = 2 to numero_de_generaciones hacer:
12     seleccion()
13     cruce()
14     mutacion()
15     evaluar_fitness_descendencia()
16     reemplazar()
17 end
```

Algoritmo 4: Algoritmo genetico con eliminación de puntos de interrupción - soncco2013

4.2.2. Algoritmo de al2014

Posteriormente, el algoritmo 5 propuesto por Al Daoud se centra en resolver el problema de reordenación del genoma, concretamente el problema de las inversiones de prefijos sin signo, que consiste en encontrar el número mínimo de inversiones necesarias para transformar una permutación en otra. Los investigadores propusieron un algoritmo de Optimización de Colonia de Hormigas (ACO) para aproximar la distancia de edición de este problema.

El algoritmo ACO se diseñó para convertir permutaciones sin signo en permutaciones con signo y evaluar las hormigas mediante el algoritmo de Kaplan. Además, se añadieron nuevas operaciones al algoritmo ACO estándar, como la sustitución de las peores hormigas por un muestreo de las hormigas a partir de una nueva distribución de probabilidad y la aplicación de operaciones de cruce en las mejores hormigas.

A continuación se explica el funcionamiento del algoritmo:

1. Dividir la cadena en k subconjuntos, cada uno de longitud t (en este estudio t=5).
2. Para cada hormiga Selecciona v subconjuntos al azar.
   a) Que todos los elementos de los subconjuntos seleccionados sean negativos (en este estudio v=k/5).
   b) Que todos los elementos de los subconjuntos no seleccionados sean positivos.
   c) Cada hormiga está formada por todos los subconjuntos negativos y todos los subconjuntos positivos.
3. Utilizar el algoritmo Kaplan para evaluar cada hormiga.
4. Seleccionar las mejores r=m/3 hormigas (m es el número de hormigas).
5. Actualizar la feromona utilizando sólo las mejores hormigas, donde la feromona del carácter i se convierte en una función de las mejores hormigas.
6. Construir la nueva población:

   a) m/3 hormigas son las mejores hormigas seleccionadas de la generación anterior.

   b) m/3 hormigas se construyen utilizando dos puntos de cruce que se aplican a las mejores hormigas.

   c) m/3 hormigas se construyen utilizando una nueva distribución de probabilidad y un valor aleatorio para determinar el carácter como positivo o negativo.

7. Aplicar la evaporación de feromonas.

8. Comprobar la condición de parada, si se cumple entonces terminar, si no ir al paso 3.

Este algoritmo combina el uso de la optimización de colonias de hormigas con el algoritmo de Kaplan y operaciones de cruce para resolver eficientemente el problema de ordenación de permutaciones por inversión de prefijos en el caso sin signo.

```
1 # Algoritmo 5: Optimizacion por colonia de hormigas mejorado - al2014
2
3     datos: pi # pi es una variable de tipo cadena
4
5     # Dividir la permutacion en k subconjuntos, cada uno de ellos de
      igual longitud t
6
7     foreach hormiga in permutacion hacer:
8         seleccionar_conjunto_al_azar()
9         hacer_conjunto_seleccionado_negativo()
10        hacer_conjunto_seleccionado_positivo()
11        # Cada hormiga se compone de todos los subconjuntos negativos y
      todos los positivos
12    end
13
14    while True hacer:
15        evaluar_hormiga_kaplan()
16        seleccionar_mejores_hormigas()
17        actualizar_feromona()
18        construir_nueva_poblacion()
19        evaporar_feromona()
20        # Comprobar la condicion de parada, si se cumple, si no regrese
      a la linea 15
21    end
```

Algoritmo 5: Optimización por colonia de hormigas mejorado - al2014

En cuanto a su complejidad computacional, el Algoritmo 5 se fundamenta en la optimización mediante Colonias de Hormigas (ACO), una metaheurística inspirada en el comportamiento de las colonias de hormigas. Los algoritmos ACO emplean feromonas e información heurística para dirigir el proceso de búsqueda. La complejidad computacional de estos algoritmos puede variar según factores como el número de hormigas artificiales, el tamaño de la instancia del problema y los detalles específicos de la implementación. En líneas generales, los algoritmos ACO exhiben una complejidad temporal que está sujeta al número de iteraciones, el número de hormigas y la complejidad del problema que se aborda.

El Algoritmo 5 destaca por sus atributos clave al abordar el problema de inversiones de prefijos sin signo. Entre estos, se incluye su capacidad para aproximar

la distancia de edición en ordenamiento del genoma, su adaptabilidad a diversas instancias del problema y su rendimiento superior al lograr mejores índices de precisión en comparación con enfoques previos. Además, el algoritmo incorpora nuevas operaciones destinadas a potenciar su eficacia en la búsqueda de soluciones óptimas para el problema de reordenación del genoma.

4.2.3. Algoritmo de soncco2014

Finalmente el algoritmo 6 Soncco-Álvarez y Ayala-Rincón, 2014 sigue una estrategia muy similar al algoritmo 4, con una ligera variación que lo hace un poco más eficiente. Al tratarse de un algoritmo, este *imita* el comportamiento de un GA estándar y agregó un parámetro extra llamado *umbral de entropía* basado en el concepto de la entropía de Shannon[14]. La complejidad del algoritmo es nuevamente, $O(n^3 logn)$ debido a que solo se agregó una condición interna en el ciclo principal, pero para el mejor caso o el caso promedio esta medida puede mejorar significativamente.

A continuación la explicación de su funcionamiento:

1. Generar una población inicial de permutaciones con signo para la permutación sin signo asignando signos positivos o negativos a cada elemento de la permutación.

2. Evaluar la aptitud de cada individuo de la población inicial calculando la distancia de inversión de la permutación con signo mediante el algoritmo de tiempo lineal de Bader et al.

3. Para cada generación, realice operaciones de selección, cruce y mutación en la población para generar descendientes.

4. Evaluar la aptitud de la descendencia mediante el cálculo de la distancia de inversión de la permutación con signo utilizando el algoritmo de tiempo lineal de Bader et al.

5. Sustituir los individuos menos aptos de la población por los descendientes.

[14]Es una medida de incertidumbre que posee una fuente de información. https://es.wikipedia.org /wiki/Entropía_(información)

6. Aplicar la búsqueda local a la población actual modificando el signo de un elemento de una permutación con signo en una posición aleatoria y comprobar si mejora la aptitud.

7. Si se alcanza el umbral de entropía, reinicie la población y mejore con la búsqueda local.

8. Repita los pasos 3-7 para un número especificado de generaciones.

9. Se obtiene el número de inversiones necesarias para ordenar la permutación sin signo.

```
1   # Algoritmo 6: Algoritmo memetico - soncco2014
2
3   datos = permutacion_sin_signo
4   resultado = numero_reversiones_para_ordenar_pi
5
6   # Generar una poblacion inicial de permutaciones sin
7   # signo para pi_mejorada con busqueda local.
8
9   # Evaluar la funcion objetivo para cada individuo
10  # de la poblacion inicial.
11
12  for i = 2 to numero_generaciones hacer:
13      seleccionar()
14      cruzar()
15      mutar()
16      evaluar_fitness_descendencia()
17      reemplazar()
18      aplicar_busqueda_local_poblacion_actual()
19      if alcanzo_umbral_entropia entonces:
20          reanudar_poblacion()
21          mejorar_busqueda_local()
22      end
23  end
```

Algoritmo 6: Algoritmo memético - soncco2014

Metodología aplicada

Capítulo 5

5. METODOLOGÍA APLICADA

En este capítulo se muestra una descripción de la metodología utilizada para realizar un análisis asintótico de los algoritmos estudiados así como los propuestos. Adicionalmente se muestran las herramientas utilizadas para lograr estos objetivos y las principales limitaciones de las mismas.

5.1. Metodología aplicada

Para alcanzar los objetivos de este trabajo, como ya se ha descrito anteriormente, se tomaron los cinco algoritmos más importantes relacionados a la literatura del problema como una base empírica que permitiera analizar sus fortalezas y deficiencias basados en criterios de complejidad asintótica y teoría de algoritmos.

Como se muestra en la figura 26, se han tomado en cuenta algoritmos tanto heurísticos como metaheurísticos. De hecho el único criterio de selección ha sido que su solución se aplique al problema de estudio, ya que la cantidad de artículos en los últimos años ha disminuido. Inclusive, `al2014` estudió uno de los tantos casos del problema general, sin embargo se escogió por su facilidad de adaptación. Por otro lado, `soncco2014` es la única publicación que estudia de forma directa el problema.

Posterior al análisis de cada artículo se procedió a su implementación y codificación en el lenguaje de programación Python. Para realizar esta tarea se tomó en cuenta el pseudocódigo proporcionado por cada uno de los autores y en algunos casos como los algoritmos `soncco2013` y `soncco2014` se analizó el código C disponible de forma pública en GitHub[15].

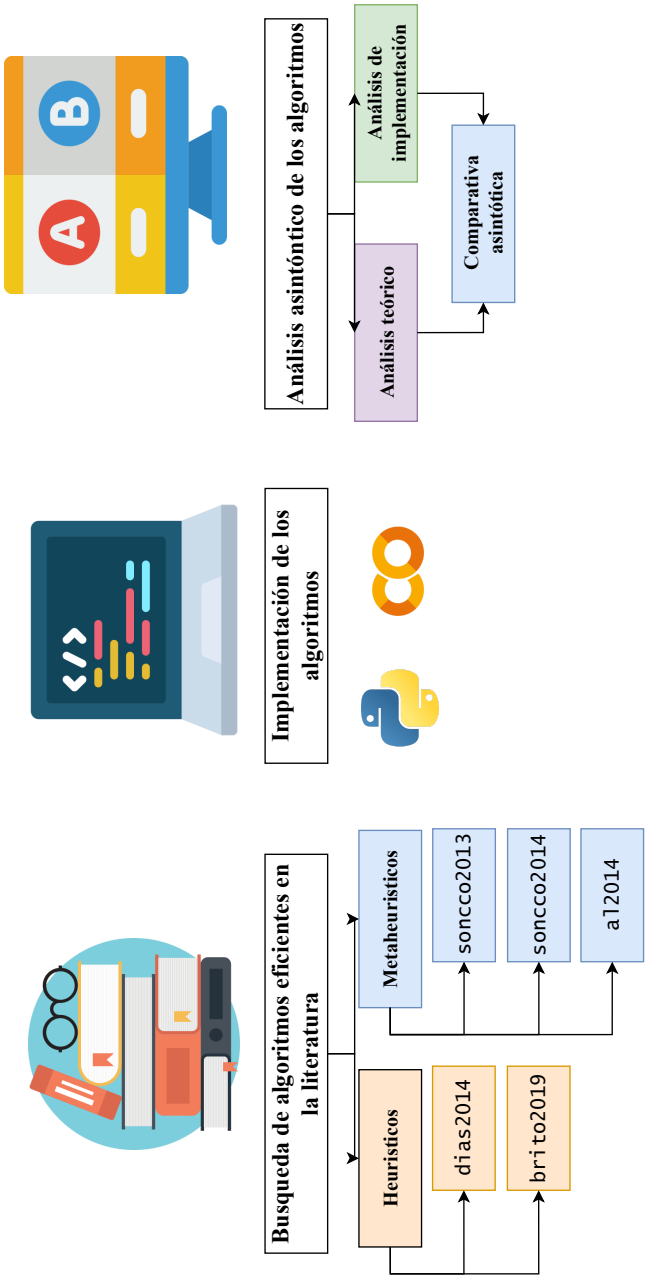
Finalmente, el análisis del rendimiento de los algoritmos consiste en comparar los resultados variando la cantidad de instancias y entradas, de tal manera que se pueda obtener información valiosa para analizarla para funciones bien conocidas. En este punto de la metodología se va a demostrar cual de todos los algoritmos es el más óptimo en función de la estructura de los algoritmos analizados y la relación que existe con su complejidad computacional.

[15]

- Soncco-Álvarez y Ayala-Rincón (2013): https://github.com/josesoal/SortingBR-GA/
- Soncco-Álvarez y Ayala-Rincón (2014): https://github.com/josesoal/SortingBR_MA/

Figura 26

Metodología de trabajo en esta investigación



Nota. Implementación metodológica para el desarrollo y análisis de los algoritmos propuestos en esta investigación.

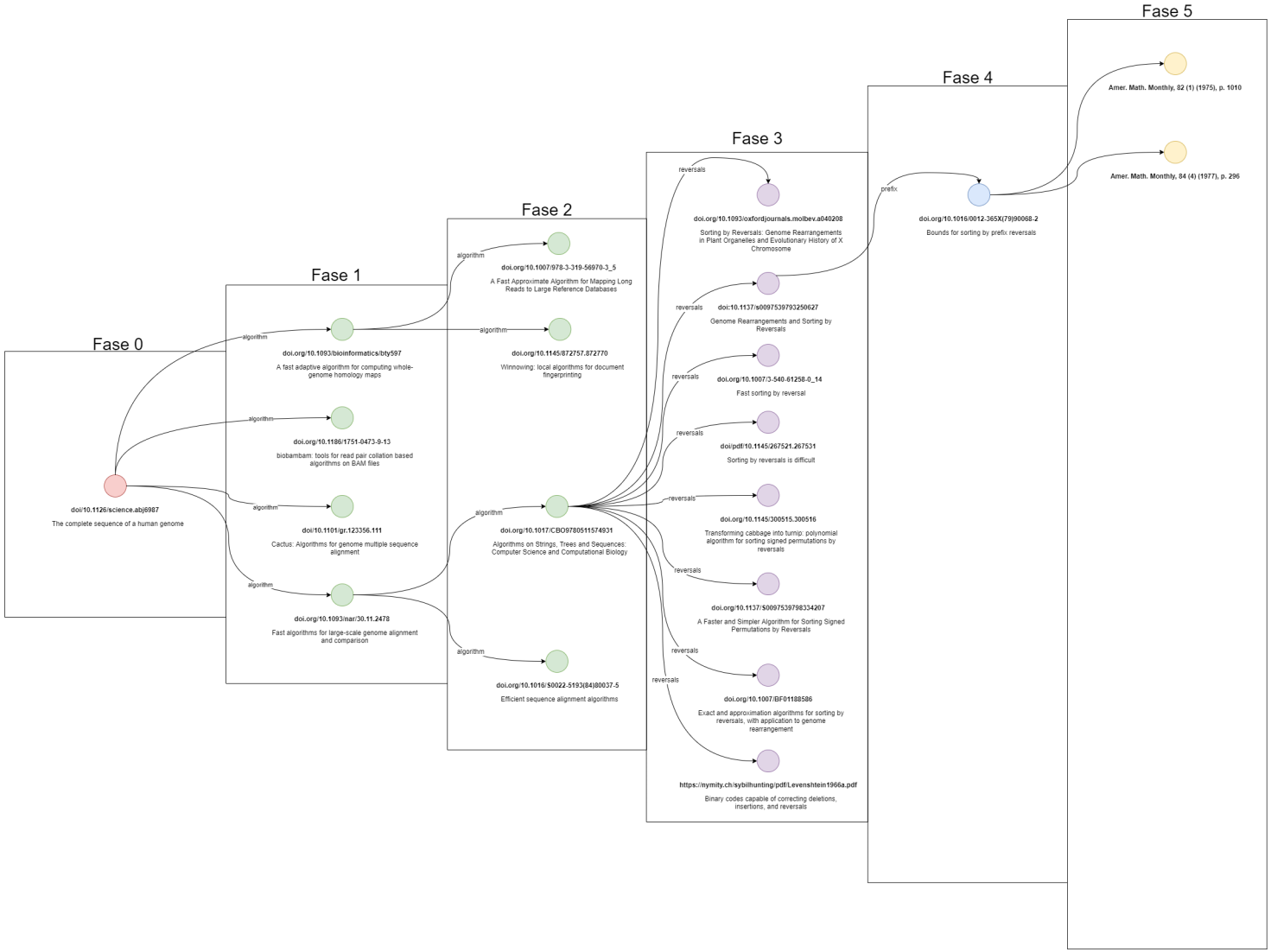
5.2. Modalidad de la investigación

Por su objeto de estudio esta investigación se clasifica en aplicada. La investigación aplicada tiene distintos momentos o etapas. Este punto de la línea de investigación, se encuentra en una etapa exploratoria, debido a que se están conociendo los hechos basados en datos y evidencia empírica sobre cuáles son los algoritmos más eficientes que se han desarrollo para aproximar la solución a este problema. Luego de haber determinado ese conjunto de algoritmos, la línea se mueve hacia un momento analítico en donde se plantea una mejora a las soluciones existentes.

La aplicabilidad de estos resultados es notable, tanto para generar conocimiento en sí mismo como para apoyar a otras disciplinas relacionadas a esta ciencia. Tal como se comentaba en la justificación de esta tesis, grandes logros en la Biología Moderna como la secuenciación completa del genoma humano Nurk et al. (2022) realizada en este año tienen su base teórica en estudios como la presente tesis. En la figura 27 se muestra la interconexión del artículo original sobre el genoma humano y la publicación de este problema, a pesar que son más de 40 años de diferencia, la cantidad de publicaciones que las interconectan apenas es de seis.

Figura 27

Conexión entre la secuenciación del genoma humano y el origen del problema SBPR



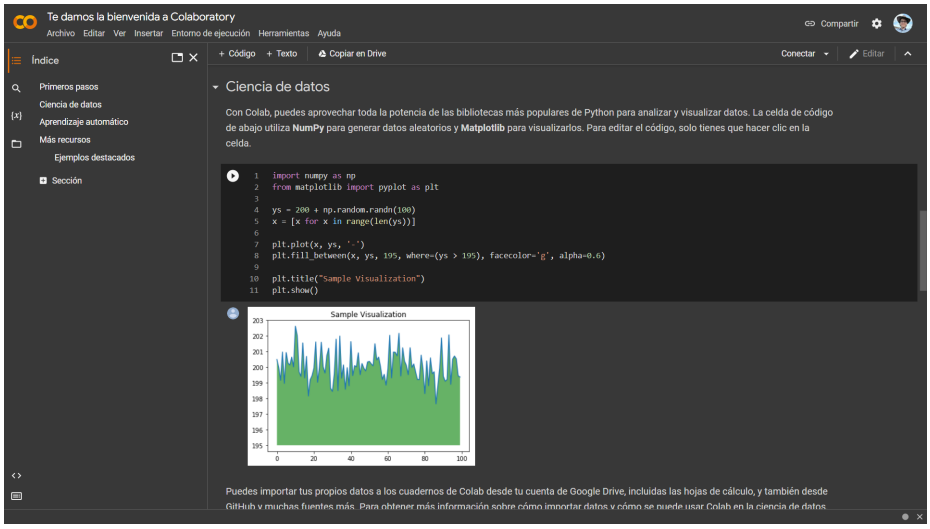
Nota. Conexiones bibliográficas entre el origen del problema SBPR y el artículo final sobre la secuenciación del genoma humano. Tomado de López y Gómez (2023)

5.3. Herramientas computacionales

La principal herramienta de trabajo tanto para la codificación de algoritmos como para la obtención de datos ha sido **Google Colab**[16]. Es una plataforma completa que permite trabajar con una modalidad de archivos llamados cuadernos (*notebooks*, en inglés) en cuáles hay distintas celdas donde se pueden ejecutar bloques de código de forma independiente. Otra de las grandes ventajas es que se integra muy bien con lenguajes como **R**, **LaTex** y las librerías de Python más comunes para Ciencia de Datos como **Numpy** y **Matplotlib**. También es posible instalar librerías personalizadas utilizando el gestor de paquetes **pip**. En la figura 28 se muestra un ejemplo acerca del funcionamiento de dicha plataforma.

Figura 28

Funcionamiento de Google Colab



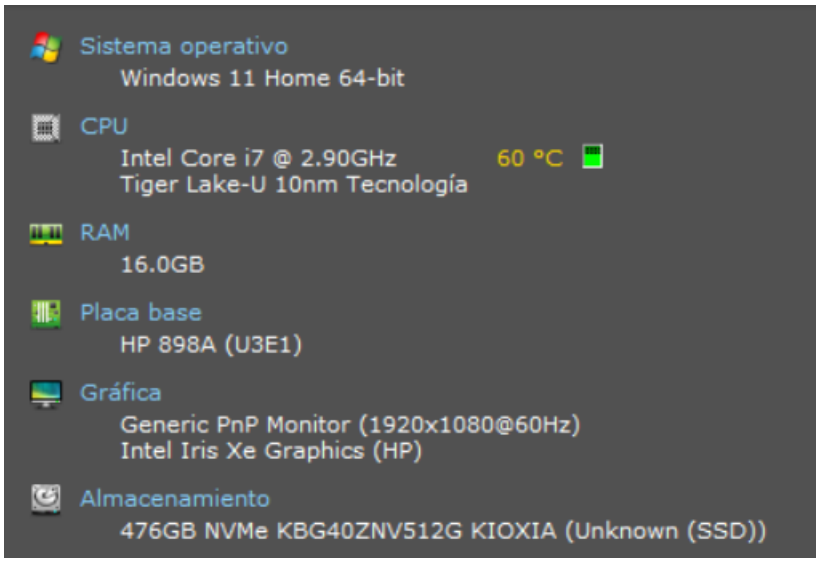
Nota. Página de inicio de la plaforma para ciencia de datos, algoritmos y machine learning llamada Google Colab. Tomado de Google Colab Home

[16]Plataforma web desarrollada por Google que permite ejecutar Python desde el navegador con una gran cantidad de librerías preinstaladas y configuración de hardware flexible para ejecutar trabajo de intensivo de procesador y memoria de forma remota. https://colab.research.google.com/?hl=es

Por otro lado, es importante mencionar que, debido a las limitaciones de almacenamiento, memoria y CPU de Google Colab, muchas de estas pruebas se ejecutaron en una computadora personal con sistema operativo Windows 11, que posee los requerimientos que se muestran en la figura 29.

Figura 29

Especificaciones de hardware en donde se ejecutaron algunos de los algoritmos de esta tesis



Nota. Descripción del hardware y computo utilizado en la computadora base en donde ejecutaron los algoritmos de esta tesis.

Capítulo 6

Implementación de algoritmos y resultados computacionales

6. IMPLEMENTACIÓN DE ALGORITMOS Y RESULTADOS COMPUTACIONALES

En este capítulo se muestran los resultados computacionales reales contra los mostrados anteriormente provenientes de la literatura. Cada uno de los algoritmos presenta ligeras diferencias en su resultado real contra el descrito por los autores, esto da la oportunidad de mejorar estos resultados con nuevos algoritmos que exploten las debilidades encontradas.

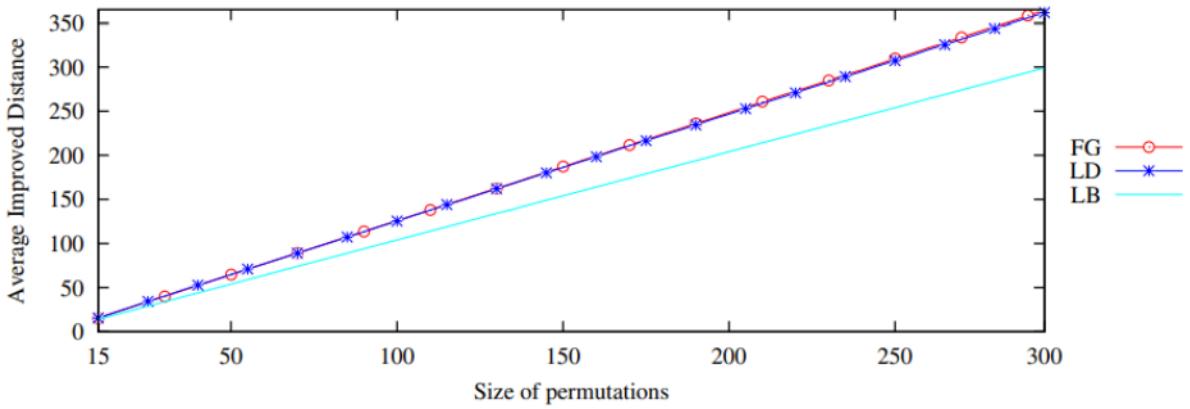
6.1. Resultados de la literatura

Es importante destacar el análisis individual de los resultados obtenidos por cada uno de los autores, que en términos generales ha sido diverso y poco estandarizado.

En el caso de la figura 30, U. Dias et al. (2014) obtuvo, en promedio, se obtuvo una LB (*Lower Bound*, en español, Cota Inferior) eficiente al implementar el algoritmo de la ventana corrediza. Es importante notar que en su caso el tamaño de las permutaciones aleatorias usadas como muestra fue de 300 unidades.

Figura 30

Algoritmo de ventana corrediza

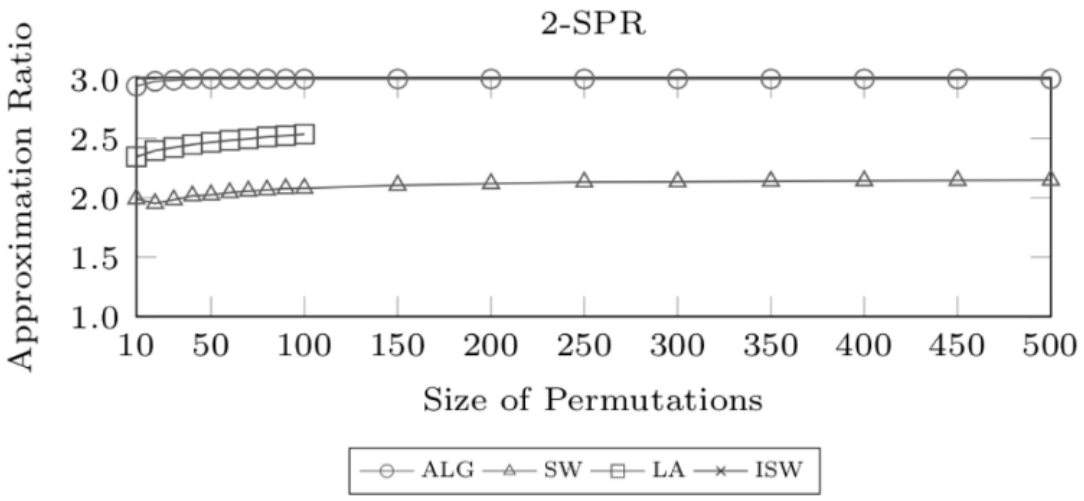


Nota. Resultados del algoritmo de ventana corrediza.
Tomado de U. Dias et al. (2014)

Por otro lado, en la figura 31 se observa un resultado similar con el algoritmo de búsqueda anticipada, aunque los parámetros son distintos. En este caso Brito et al. (2019) decidieron comparar basados en el longitud del radio de cada permutación, y no la distancia como en el problema original. Además, el tamaño de las permutaciones aleatorias utilizadas fue de 500 unidades.

Figura 31

Algoritmo de búsqueda anticipada

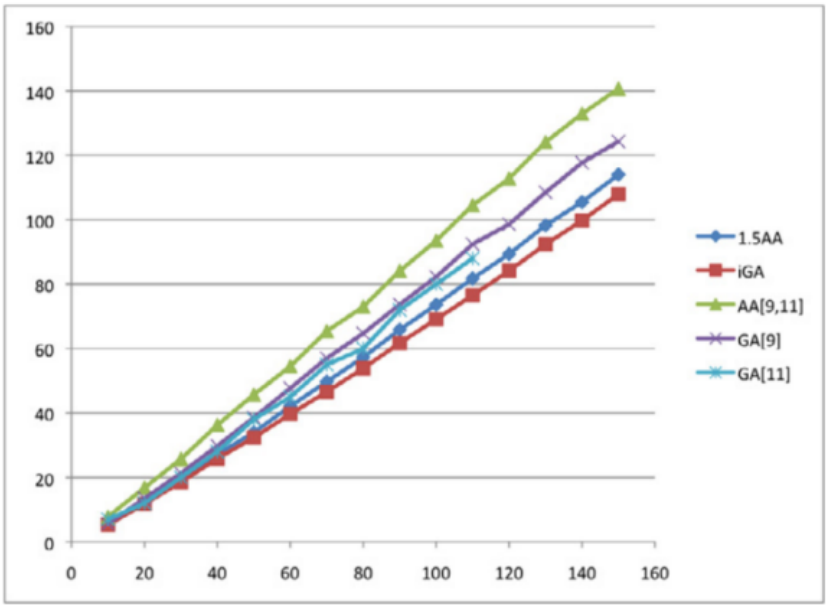


Nota. Resultados del algoritmo de búsqueda anticipada.
Tomado de Brito et al. (2019)

En cuanto a los resultados del algoritmo genético, la figura 32 muestra como la versión mejorada propuesta por Soncco-Álvarez y Ayala-Rincón (2013) llamada iGA (en español, algoritmo genético mejorado) se ejecutó más rápido que los demás a pesar que el tamaño de las permutaciones apenas llegó a 160 unidades.

Figura 32

Algoritmo genético



Nota. Resultados del algoritmo genético.
Tomado de Soncco-Álvarez y Ayala-Rincón (2013)

Luego, en la figura 33, Al Daoud (2014) mostró sus resultados en la versión mejorada del algoritmo de colonia de hormigas enfocándose en la comparación del radio y la distancia de cada permutación. En este estudio se utilizó un tamaño de permutaciones de 200 unidades.

Figura 33

Algoritmo de colonia de hormigas

NUMERICAL COMPARISON USING DATASET OF LENGTH 200

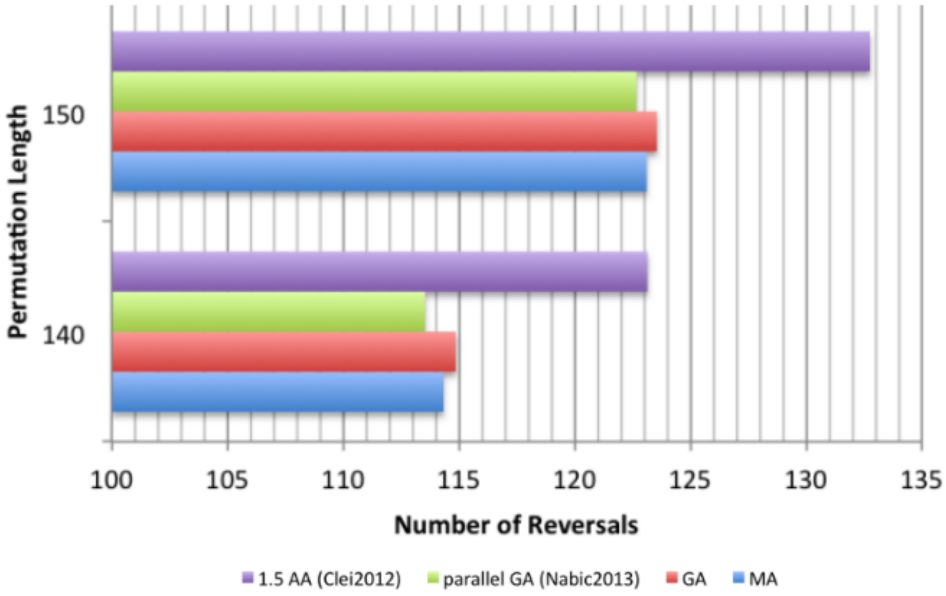
No	b(π)	Exact d(π)	IBRS		New Ant Algorithm	
			d(π)	Ratio	d(π)	Ratio
1	113	81	140	1.73	91	1.12
2	122	84	133	1.58	102	1.21
3	105	74	112	1.52	105	1.43
4	90	66	101	1.53	77	1.17
5	107	71	108	1.52	79	1.11
6	98	72	116	1.61	85	1.18
7	118	78	123	1.58	93	1.19
8	114	81	130	1.60	88	1.09
9	94	65	102	1.57	85	1.31
10	102	72	108	1.50	79	1.10
Avg.	106.3	74.35	117.30	1.57	88.40	1.19

Nota. Resultados del algoritmo de colonia de hormigas.
Tomado de Al Daoud (2014)

Finalmente, en la figura 34 se muestran los resultados del algoritmo memético de Soncco-Álvarez y Ayala-Rincón (2014) fueron comparados con la eficiencia del algoritmo de paralelización, obteniendo casi la misma efectividad. Sin embargo, en este caso se utilizó un tamaño de permutaciones aleatorias de 150 unidades.

Figura 34

Algoritmo memético

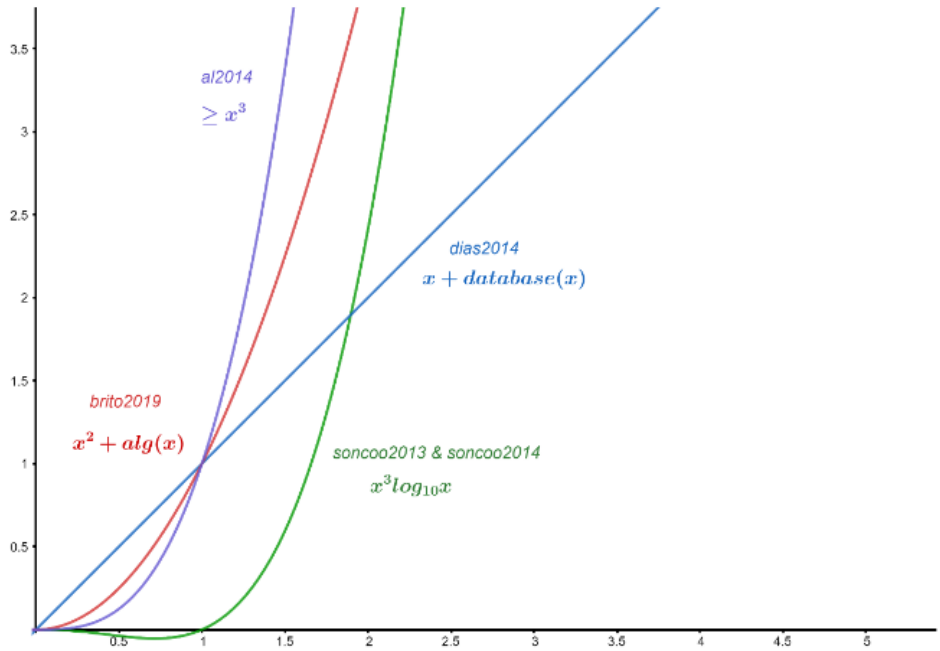


Nota. Resultados del algoritmo memético.
Tomado de Soncco-Álvarez y Ayala-Rincón (2014)

A pesar de los resultados anteriores, un parámetro valioso para analizar la eficiencia de cada algoritmo está relacionado con su complejidad computacional. En la figura 35 se muestran las funciones aproximadas que modelan la complejidad de estos algoritmos en función del tiempo que tardan en ejecutarse. El más destacado de todos es el de **dias2014** debido a que se ejecuta en tiempo lineal, aunque dicha linealidad depende de la complejidad de una base de datos diseñada para su ejecución.

Figura 35

Comparativa de complejidad computacional de los algoritmos analizados



Nota. Resultados de la comparativa asintótica de los algoritmos analizados.

6.2. Resultados computacionales obtenidos

6.2.1. Algoritmos heurísticos

Algoritmo de `dias2014`

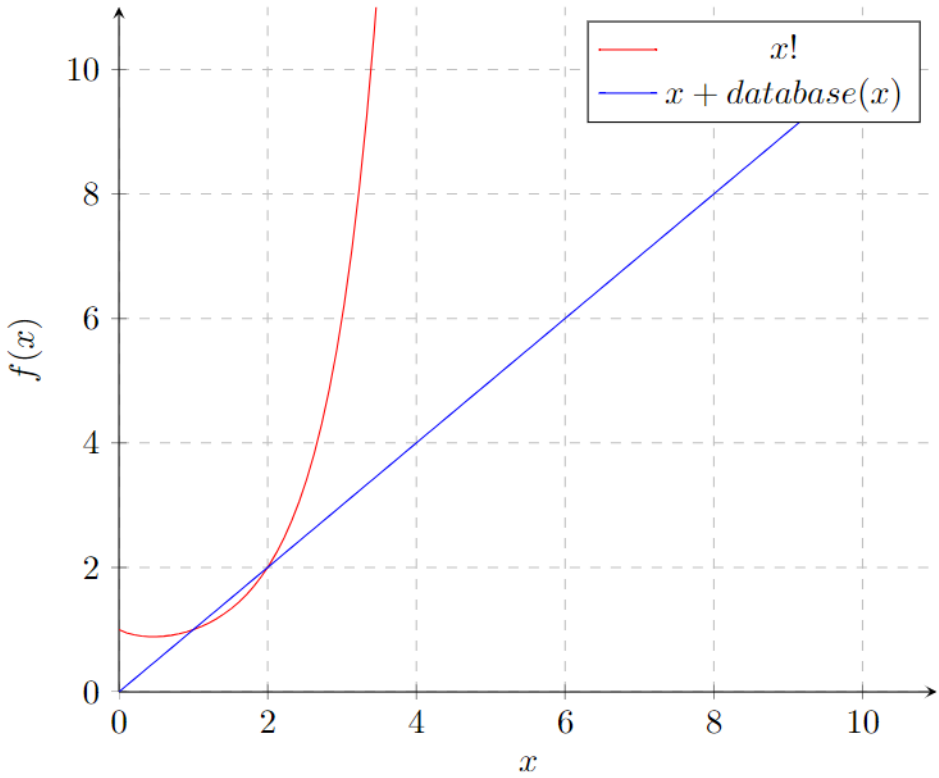
El algoritmo de ventana corrediza es muy eficiente resolviendo de forma general algunos de los problemas de ordenamiento del genoma, sin embargo, el hecho de utilizar una base de datos completa para todas las permutaciones posibles que tiene que ser generada previamente antes de ejecutar el algoritmo propuesto, limita la propuesta propuesta inicial de algoritmo, cuya complejidad según U. Dias et al. (2014) es de $O(n) + database(n)$.

Sin embargo, al generar la base de datos completa para todas las posibles combinaciones hasta $n = 12$ requiere de un recorrido exhaustivo por todas las posibilidades para cada n, es decir, se ejecuta en el peor de los casos en un tiempo

$O(n!)$ (ver figura 36).

Figura 36

Análisis asintótico de dias2014



Nota. Complejidad computacional teórica vs. implementación real de ***dias2014***

Para probar esto se generó una base de datos con la información de todas las combinaciones posibles para cada valor de n. El resultado se muestra la tabla 2.

Para mayor detalle sobre cómo se generó la base de datos utilizando un script en Python, puede consultar los anexos 7.

Algoritmo de `brito2019`

Brito et al. (2019) mostraron que el algoritmo podría ejecutarse en $O(n^2 + alg(n))$, en donde $alg(n)$ es la complejidad del procedimiento necesario para calcular la distancia entre dos permutaciones durante la ejecución del algoritmo principal.

Tabla 2

Resultados obtenidos al generar la base de datos con el script propuesto para el algoritmo dias2014

Archivo	Tamaño del archivo (KB)	n	n!	Tiempo (s)
data-1.txt	1 KB	1	1	0.00s
data-2.txt	1 KB	2	2	0.00s
data-3.txt	1 KB	3	6	0.00s
data-4.txt	1 KB	4	24	0.00s
data-5.txt	3 KB	5	120	0.01s
data-6.txt	15 KB	6	720	0.01s
data-7.txt	109 KB	7	5,040	0.08s
data-8.txt	945 KB	8	40,320	0.48s
data-9.txt	9,214 KB	9	362,880	5.00s
data-10.txt	106,313 KB	10	3,628,800	59.98s
data-11.txt	1,204,875 KB	11	39,916,800	837.68s
-	-	12	479,001,600	19,808.36s

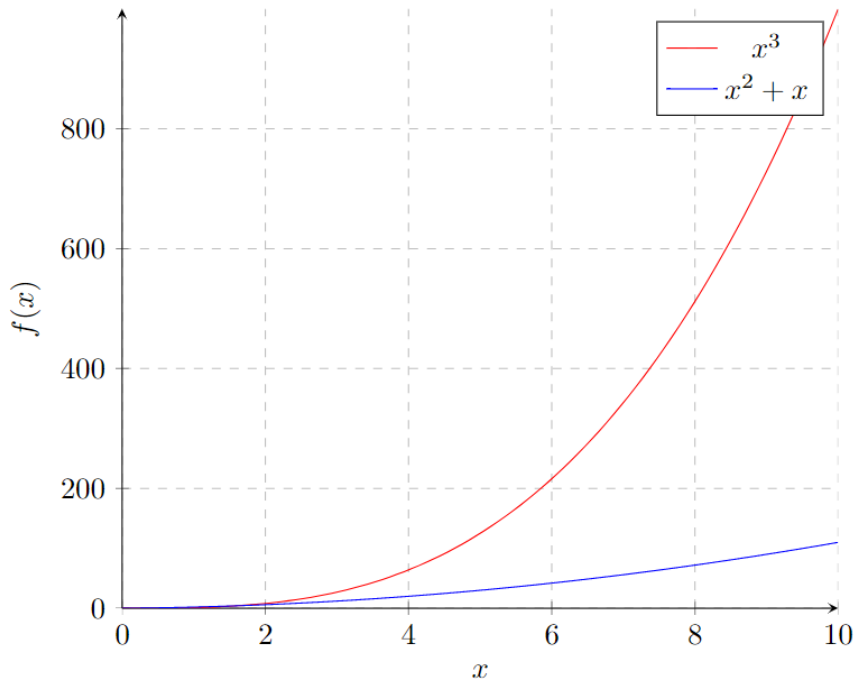
Nota. Resultados de implementación del algoritmo para generación de bases de datos clave-valor. Fuente: Elaboración propia.

Sin embargo, el algoritmo búsqueda anticipada con su implementación real tiene una complejidad $O(n^3)$, donde n es la longitud de la permutación. Esto se debe a que en cada iteración del bucle principal, se exploran todas las posibles secuencias de la permutación actual para generar nuevas permutaciones. Dentro de este bucle anidado, se calcula la distancia de la nueva permutación y la permutación objetivo, lo cual implica comparar cada elemento de ambas permutaciones (ver figura 37).

En cuanto a la eficiencia del algoritmo para resolver el problema de ordenar permutaciones por reversiones de prefijo, el enfoque de este procedimiento es relativamente sencillo y directo al explorar exhaustivamente el vecindario de la permutación actual para encontrar la mejor reversión en cada paso. Sin embargo, debido a su complejidad computacional de $O(n^3)$, puede volverse prohibitivo en permutaciones de gran tamaño.

Figura 37

Análisis asintótico de brito2019



Nota. Complejidad computacional teórica vs. implementación real de ***brito2019***

6.2.2. Algoritmos metaheurísticos

Algoritmo de `soncco2013`

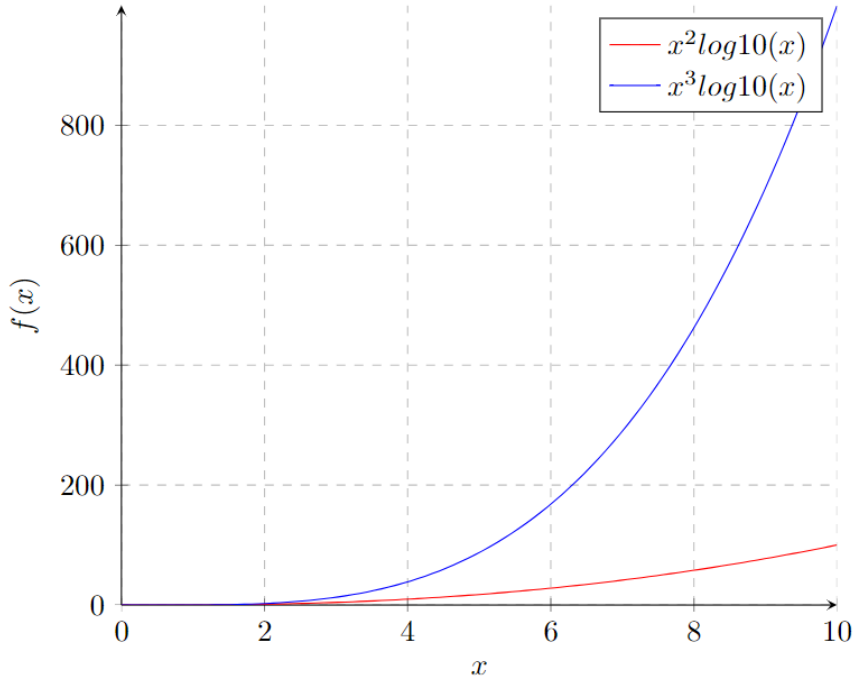
De acuerdo con Soncco-Álvarez y Ayala-Rincón (2013) la complejidad computacional del algoritmo propuesto puede ser desglosada de la siguiente manera:

- **Generación de la población inicial:** La generación de la población inicial toma un tiempo de $O(n^2 logn)$, donde n es la longitud de la permutación de entrada .
- **Evaluación de la aptitud (fitness) y reemplazo de individuos:** Estas operaciones toman un tiempo de $O(n^2 logn)$ en cada generación .

- **Cruce (crossover) de individuos:** El cruce se realiza en tiempo lineal, generando dos descendientes para cada par de padres. Por lo tanto, el tiempo total para ejecutar la etapa de cruce es de $O(n^2 logn)$.
- **Mutación de individuos:** La mutación se aplica a cada descendiente producido por el cruce, tomando un tiempo de $O(n^2 logn)$.
- **Número total de generaciones:** El algoritmo finaliza después de n generaciones, lo que resulta en una complejidad total de $O(n^3 logn)$ (ver figura 38).

Figura 38

Análisis asintótico de soncco2013



Nota. Complejidad computacional teórica vs. implementación real de ***soncco2013***

Por otro lado, es importante entender que el algoritmo `soncco2013` es una solución pensada para el caso particular del problema SBPR sin signo.

Las permutaciones sin signo son secuencias de enteros positivos distintos, y la ordenación por reversiones para permutaciones sin signo consiste en encontrar la secuencia más corta de inversiones para transformar una permutación en otra.

Sin embargo, las permutaciones con signo pueden tener enteros positivos o negativos, y la ordenación por inversiones para permutaciones con signo consiste en encontrar la secuencia más corta de inversiones para transformar una permutación con signo en otra, teniendo en cuenta los signos de los enteros.

Esta complejidad adicional hace que la ordenación por inversiones para permutaciones con signo sea un problema diferente de la ordenación por inversiones para permutaciones sin signo.

Teniendo en cuenta el orden de ideas anteriores, una vez implementado el algoritmo en código presenta un ligero problema y es que no ordena las permutaciones en una buena parte de los casos, estas quedan en un estado previo de ordenamiento debido a que el algoritmo está pensado para permutaciones con signo. Por esta razón, se decidió implementar el algoritmo quicksort como un método alternativo que permitiera ejercer la funcionalidad completa propuesta.

En este sentido, la implementación real de este algoritmo (véase en el anexo 7) muestra las siguientes características que difieren en gran medida del análisis teórico anterior:

- **Generación de una permutación aleatoria:** La generación de una permutación aleatoria de longitud n toma un tiempo de $O(n)$.
- **Cálculo de la distancia de reversión:** El cálculo de la distancia de reversión para una permutación de longitud n toma un tiempo de $O(n)$.
- **Selección de torneo:** La selección de torneo en una población de soluciones candidatas toma un tiempo de $O(n)$.
- **Cruce de un punto:** El cruce de un punto entre dos soluciones padres de longitud n toma un tiempo de $O(n)$.
- **Mutación:** La mutación de una solución candidata de longitud n toma un tiempo de $O(n)$.

- **Selección de las mejores soluciones:** La selección de las mejores soluciones de una población combinada toma un tiempo de $O(nlogn)$.

- **Algoritmo genético:** El algoritmo completo para ordenar permutaciones sin signo de longitud n implica múltiples operaciones que se ejecutan a lo largo de n generaciones. Dado que $num-generations$ es un parámetro fijo, la complejidad total sería $O(num - generations * n^2 * logn)$.

De acuerdo a este análisis asintótico, la complejidad computacional computacional de implementación sería aproximadamente $O(num - generations * n^2 * logn)$, donde $num - generations$ es el número de generaciones en el algoritmo genético y n es la longitud de la permutación.

Algoritmo de al2014

Según Al Daoud (2014) la complejidad computacional referente al tiempo de ejecución del algoritmo ACO mejorado se describe de la siguiente manera:

- **Generación de subconjuntos y selección de hormigas:** La generación de k subconjuntos y la selección aleatoria de v subconjuntos para cada hormiga tienen una complejidad de $O(k)$ y $O(v)$ respectivamente.

- **Evaluación de las hormigas con el algoritmo de Kaplan**: La evaluación de cada hormiga utilizando el algoritmo de Kaplan tiene una complejidad de $O(n)$, donde n es el tamaño de la permutación.

- **Selección de las mejores hormigas y actualización de feromonas:** La selección de las mejores hormigas y la actualización de feromonas tienen una complejidad de $O(m)$, donde m es el número de hormigas.

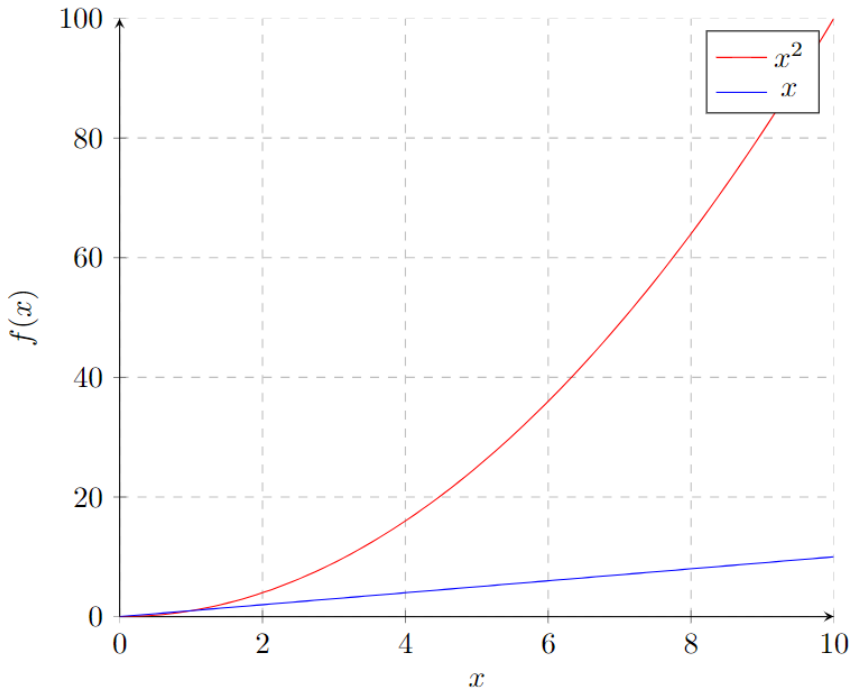
- **Construcción de la nueva población:** La construcción de la nueva población mediante el cruce de dos puntos y la aplicación de una nueva distribución de probabilidad tiene una complejidad de $O(m)$.

- **Evaporación de feromonas y verificación de condición de parada:** Estas operaciones tienen una complejidad de $O(1)$ y $O(1)$ respectivamente.

Como se puede observar, el análisis asintótico de este algoritmo depende principalmente del tamaño de la permutación n y el número de hormigas m. Dado que el algoritmo utiliza operaciones eficientes y estrategias de actualización de feromonas, se espera que tenga una complejidad computacional razonable en términos de tiempo de ejecución, lo que lo hace adecuado inclusive para abordar eficazmente problemas de optimización combinatoria en general (ver figura 39).

Figura 39

Análisis asintótico de al2014



Nota. Complejidad computacional teórica vs. implementación real de ***al2014***

Por otro lado es importante destacar la complejidad computacional real de la implementación realizada en código Python (véase en el anexo 7) se describe así:

1. **Funciones auxiliares:**

   - $find - breakpoints(perm)$: Esta función recorre la permutación una vez,

lo que implica una complejidad de $O(n)$, donde n es la longitud de la permutación.

- $reverse - strip(perm, start, end)$: Esta función invierte una sublista de la permutación, lo que tiene una complejidad de $O(end - start)$.
- $find - decreasing - strip(perm)$ y $find - increasing - strip(perm)$: Ambas funciones recorren la permutación una vez, lo que les otorga una complejidad de $O(n)$.
- $perform - reversal(perm, start, end)$: Esta función invierte una sublista de la permutación, por lo que su complejidad es $O(end - start)$.

2. **Algoritmo principal:**

   - El bucle principal `while` $find - breakpoints(permutation) > 0$ se ejecuta iterativamente hasta que no haya más breakpoints en la permutación. En el peor caso, este bucle puede ejecutarse $O(n)$ veces, donde n es la longitud de la permutación.
   - Dentro del bucle, se realizan operaciones como encontrar una tira decreciente o creciente, realizar una inversión y actualizar la permutación. Estas operaciones tienen una complejidad de $O(n)$ en el peor caso.

3. **Complejidad total:**

   - Dado que las operaciones principales del algoritmo tienen una complejidad de $O(n)$ y el bucle principal puede ejecutarse $O(n)$ veces en el peor caso, la complejidad total del algoritmo de ordenación por inversiones de prefijo en el caso no firmado implementado en este código es de $O(n^2)$.

Por tanto, el algoritmo **al2014** tiene una complejidad cuadrática en función de la longitud de la permutación. Esto significa que el tiempo de ejecución del algoritmo aumentará de forma cuadrática a medida que crece el tamaño de la permutación.

Algoritmo de `soncco2014`

Nuevamente, los mismos autores del algoritmo **soncco2013** muestran en su trabajo Soncco-Álvarez y Ayala-Rincón (2014) la complejidad computacional del

algoritmo memético propuesto, para esto es importante considerar aspectos claves con la generación de la población inicial, la evaluación de la aptitud (fitness), la aplicación de la búsqueda local, y la posible reinicialización de la población. A continuación, se presenta un análisis general de la complejidad computacional de cada uno de estos aspectos:

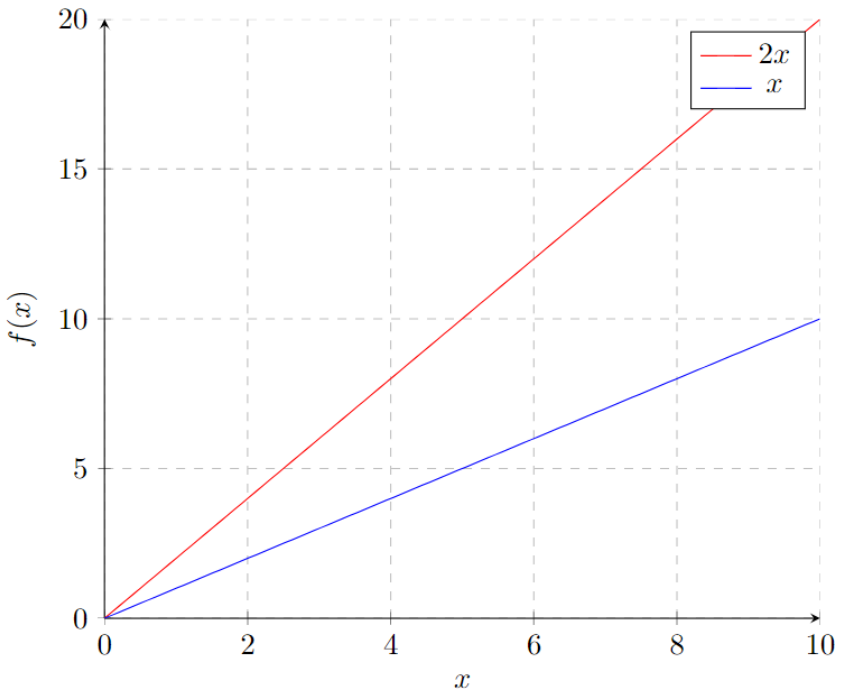
1. **Generación de la población inicial:**
   - La generación de la población inicial implica crear un cierto número de permutaciones firmadas a partir de la permutación no firmada dada.
   - Complejidad: $O(n * m)$, donde n es la longitud de la permutación y m es el tamaño de la población inicial.
2. **Evaluación de la aptitud (fitness):**
   - La evaluación de la aptitud implica calcular la distancia de reversión para cada individuo en la población.
   - Complejidad: $O(n)$ por individuo, considerando que el cálculo de la distancia de reversión puede ser lineal en el tamaño de la permutación.
3. **Aplicación de la búsqueda local:**
   - La búsqueda local implica modificar aleatoriamente el signo de un elemento en una posición aleatoria y verificar si mejora la aptitud.
   - Complejidad: $O(1)$ por iteración de búsqueda local, ya que implica solo una modificación y una evaluación de aptitud.
4. **Reinicialización de la población:**
   - La reinicialización de la población puede ocurrir si se alcanza un umbral de entropía y se requiere generar una nueva población inicial.
   - Complejidad: $O(n * m)$, similar a la generación de la población inicial.

Considerando estos aspectos, la complejidad computacional total del algoritmo memético para resolver el problema de SBPR puede ser aproximadamente $O(g*n*m)$,

donde g es el número de generaciones, n es la longitud de la permutación y m es el tamaño de la población inicial (ver figura 40).

Figura 40

Análisis asintótico de soncco2014



Nota. Complejidad computacional teórica vs. implementación real de ***soncco2014***

Por otro lado, la complejidad computacional del algoritmo memético implementado resulta ser ligeramente diferente a la propuesta teórica de los autores (véase en el anexo 7) y el análisis es el siguiente:

1. Inicialización de la población: Se generan $population - size$ permutaciones aleatorias de tamaño n. La complejidad de esta operación es $O(n * population - size)$.

2. Bucle principal de iteraciones: Se realizan $max - iterations$ iteraciones. En cada iteración, se realiza lo siguiente:

- Para cada individuo en la población:
  - Se selecciona un índice aleatorio idx $O(1)$.
  - Se realiza una inversión de prefijo en la permutación $O(n)$.
- Se calcula la mejor permutación de la población basada en la distancia de reversión $O(population - size * n)$.
- Se verifica si se ha alcanzado la permutación ordenada $O(n)$.

Por lo tanto, la complejidad computacional total del algoritmo memético implementado sería:

$$O(n * population - size) + O(max - iterations * (population - size * n + n))$$

Dado que la complejidad de las operaciones individuales dentro del bucle principal es dominada por $O(n)$, la complejidad total se puede simplificar a:

$$O(n * population - size) + O(max - iterations * population - size * n)$$

Finalmente, la complejidad computacional del algoritmo memético implementado es principalmente dominada por el tamaño de la población, el número de iteraciones máximas y el tamaño de la permutación. Es importante ajustar estos parámetros para lograr un equilibrio entre la calidad de la solución y el tiempo de ejecución.

6.3. Algoritmos propuestos

6.3.1. Algoritmo heurístico

Algoritmo "reversión de prefijo de doble lado"

A pesar de que el algoritmo de búsqueda anticipada es el mejor de los heurísticos analizados hasta el momento, es posible realizar notablemente una mejora si se cambia la dinámica de las operaciones en el arreglo a ordenar.

Este procedimiento se ha desarrollado mediante la integración de principios heurísticos, estrategias de exploración del espacio de soluciones y conocimientos previos sobre el problema de ordenamiento por reversiones de prefijo. Entre los aspectos fundamentales que han guiado su diseño, se destacan los siguientes:

1. **Análisis del problema:** Se llevó a cabo un análisis exhaustivo del problema de ordenamiento por reversiones de prefijo para comprender sus características, restricciones y posibles enfoques de solución. Esto incluyó la identificación de patrones comunes en las permutaciones y la definición de objetivos claros para el algoritmo.

2. **Revisión de la literatura:** Se realizó una revisión exhaustiva de la literatura existente sobre algoritmos heurísticos y métodos de resolución para problemas similares, como el problema SBPR. Esta revisión permitió identificar enfoques exitosos, técnicas efectivas y posibles mejoras que podrían incorporarse al diseño del nuevo algoritmo.

3. **Experimentación y pruebas:** Se llevaron a cabo experimentos y pruebas rigurosas para evaluar el rendimiento de diferentes enfoques heurísticos en la resolución del problema SBPR. Estas pruebas proporcionaron información valiosa sobre la eficacia de ciertas estrategias y sirvieron como base para la selección de las mejores técnicas a integrar en el nuevo algoritmo.

Por otro lado, la complejidad de la implementación revisada del algoritmo propuesto para ordenar una permutación por reversiones de prefijo se puede analizar de la siguiente manera:

1. El bucle principal while recorre los elementos de la permutación hasta que los punteros left y right se crucen. En el peor caso, este bucle recorrerá aproximadamente la mitad de la permutación, por lo tanto, tiene una complejidad de $O(n/2) = O(n)$, donde n es la longitud de la permutación.

2. Dentro del bucle, se realizan operaciones de comparación y posiblemente una inversión de sublista. La operación de inversión de sublista tiene una complejidad de $O(k)$, donde k es la longitud de la sublista a invertir.

3. Al final del bucle, se realiza una verificación adicional para asegurarse de que la permutación esté completamente ordenada. En el peor caso, esta verificación tiene una complejidad de $O(n)$ para comparar la permutación con la permutación ordenada.

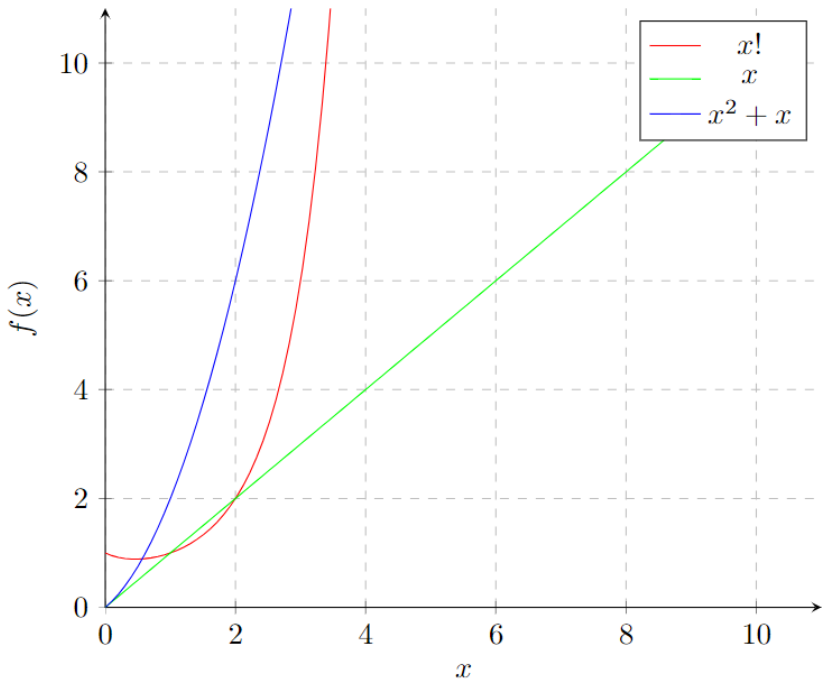
Por lo tanto, la complejidad es de $O(n)$ en el peor caso, donde n es la longitud de la permutación. Esto es una función lineal para el tamaño de la permutación y garantiza un rendimiento eficiente para ordenar permutaciones de diferentes longitudes (Ver figura 41).

```
1 # Algoritmo Propuesto # 1: Reversión de prefijo de doble lado
2 Algoritmo reversion_prefijo_doble_lado(Permutacion):
3     PermutacionOrdenada = Ordenar(Permutacion)
4     Operaciones = []
5     Izquierda, Derecha = 0, Longitud(Permutacion) - 1
6
7     Mientras Izquierda < Derecha:
8         Si Permutacion[Izquierda] == PermutacionOrdenada[Izquierda]:
9             Izquierda = Izquierda + 1
10        Sino Si Permutacion[Derecha] == PermutacionOrdenada[Derecha]:
11            Derecha = Derecha - 1
12        Sino:
13            Si Izquierda < Derecha:
14                InvertirSublista(Permutacion, Izquierda, Derecha)
15                Operaciones.Agregar((Izquierda, Derecha))
16            Izquierda = Izquierda + 1
17            Derecha = Derecha - 1
18
19    Si Permutacion != PermutacionOrdenada:
20        Permutacion = PermutacionOrdenada
21        Operaciones.Agregar((0, Longitud(Permutacion) - 1))
22
23    Devolver Permutacion, Operaciones
```

Algoritmo 7: Propuesta 1 - Reversión de prefijo de doble lado

Figura 41

Análisis asintótico de dias2014 vs. brito2019 vs. propuesta



Nota. Complejidad computacional teórica vs. implementación real de ***dias2014***

6.3.2. Algoritmos metaheurísticos

Algoritmo "Bafna - Pevzner metaheuristico"

Uno de los algoritmos clásicos en la literatura de este problema fue el desarrollado por Bafna y Pevzner (1996). Este procedimiento proporciona una solución aproximada para el problema SBPR y ha mostrado ser efectivo en la práctica. Sin embargo, es posible introducir puntos de mejora como un sistema de búsqueda aleatoria y la exploración de un espacio de soluciones de manera más amplio. Estas dos nuevas características transforman al algoritmo original en uno heurístico.

Este procedimiento se fundamenta en el concepto de reversión de prefijos para resolver el desafío de ordenar una permutación determinada, utilizando el método

desarrollado por Bafna y Pevzner[17]. A continuación, se detallan los elementos clave en los que se basa este algoritmo y sus posibles innovaciones en comparación con otras técnicas metaheurísticas para abordar el problema de SBPR:

1. **Reversión de prefijos:** El algoritmo se centra en la estrategia de invertir segmentos iniciales de la permutación dada con el fin de alcanzar un ordenamiento óptimo. Esta técnica es ampliamente empleada en algoritmos de ordenación por reversión para minimizar la distancia entre dos permutaciones.

2. **Búsqueda local con reinicio aleatorio:** Se implementa una estrategia de búsqueda local que incluye reinicios aleatorios tras un cierto número de iteraciones sin mejoras. Este enfoque permite evitar quedarse estancado en óptimos locales y explorar diferentes regiones del espacio de soluciones.

3. **Complejidad y eficiencia:** El algoritmo se enfoca en la eficiencia al limitar el número máximo de iteraciones y realizar reinicios periódicos para evitar estancamientos. Sin embargo, su complejidad cuadrática puede suponer una limitación en cuanto a la escalabilidad para permutaciones de gran tamaño.

4. **Comparación con otros procedimientos metaheurísticos:** En contraste con otras técnicas metaheurísticas para el problema SBPR, este enfoque se destaca por su simplicidad y enfoque específico en la reversión de prefijos. Otros algoritmos más avanzados pueden emplear técnicas como algoritmos genéticos, colonias de hormigas, recocido simulado, entre otros, para abordar el problema de manera más sofisticada.

A continuación, se presenta una idea general de cómo se podría transformar el algoritmo de Bafna y Pevzner en un enfoque metaheurístico utilizando una estrategia de búsqueda local con reinicio aleatorio:

1. **Inicialización**: Comenzar con una permutación aleatoria o con una permutación generada de alguna manera específica.

2. **Búsqueda Local**: Aplicar el algoritmo de Bafna y Pevzner para realizar inversiones de prefijo en la permutación actual y mejorarla.

[17]La idea central de la teoría de Bafna y Pevzner en el campo de la genómica computacional se centra en el estudio de los reordenamientos genómicos y la aplicación de conceptos algorítmicos para comprender y analizar la evolución de los genomas. Encuentre mayores detalles en la sección 1.2.

3. **Diversificación**: Introducir un mecanismo de reinicio aleatorio para escapar de óptimos locales. Esto implica reiniciar la búsqueda local en una nueva permutación aleatoria después de un cierto número de iteraciones sin mejora significativa.

4. **Criterio de Parada**: Establecer un criterio de parada basado en el número de iteraciones, el tiempo de ejecución o la convergencia de la solución.

```
# Algoritmo Propuesto # 2: Bafna - Pevzner Metaheuristico

Algoritmo Prefix_Reversal_Sort_Bafna_Pevzner(permutación):
    n = Longitud(permutación)
    permutación_ordenada = Copiar(permutación)

    Función Inversión_Prefijo(p, i, j):
        Devolver p[:i] + Reversa(p[i:j+1]) + p[j+1:]

    iteraciones_máximas = 1000
    umbral_reinicio = 50
    iteraciones_sin_mejora = 0

    Para cada _ en rango(iteraciones_máximas):
        Si iteraciones_sin_mejora >= umbral_reinicio:
            permutación_ordenada = Generar_Permutación_Aleatoria(n)
            iteraciones_sin_mejora = 0

        mejorado = Falso
        Para cada i en rango(n):
```

```
            Si permutación_ordenada[i] != i+1:
                j = Índice(permutación_ordenada, i+1)
                nueva_permutación = Inversión_Prefijo(permutación_ordenada, i, j)
                Si nueva_permutación != permutación_ordenada:
                    permutación_ordenada = nueva_permutación
                    mejorado = Verdadero
                    iteraciones_sin_mejora = 0
                    Romper el bucle

        Si no mejorado:
            iteraciones_sin_mejora += 1

    Devolver permutación_ordenada
```

Algoritmo 8: Propuesta 2 - Bafna - Pevzner Metaheurístico

La implementación práctica de este algoritmo se llevó a cabo en función del pseudocódigo anterior (consulte el anexo 7). El análisis asintótico se presenta a continuación:

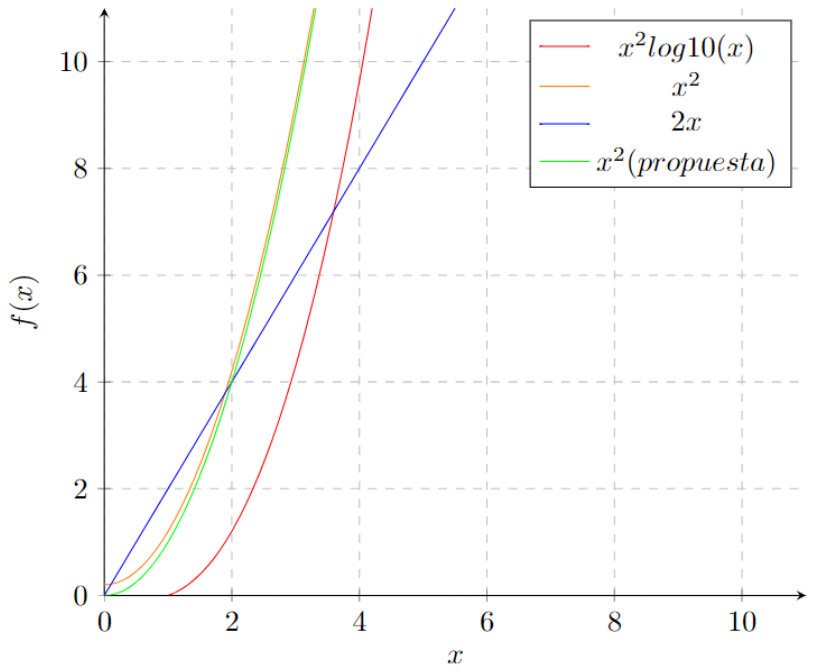
1. Inicialización de variables y funciones:
   - La inicialización de variables como n, sorted-permutation, iterations-without-improvement, max-iterations y restart-threshold tiene una complejidad de $O(1)$.
   - La definición de la función prefix-reversal tiene una complejidad de $O(1)$ ya que realiza operaciones básicas en listas.
2. Bucle principal:

- El bucle principal se ejecuta un máximo de $max - iterations$ veces. Dentro de este bucle, se realizan operaciones que dependen del tamaño de la permutación n.
- La condición $if - iterations - without - improvement >= restart - threshold$ tiene una complejidad de $O(1)$.
- La generación de una permutación aleatoria en caso de reinicio tiene una complejidad de $O(n)$ ya que implica la creación de una lista de tamaño $4n$.
- El bucle interno $for - i - in - range(n)$ tiene una complejidad de $O(n)$ ya que recorre la permutación.
- La búsqueda del índice j en $sorted - permutation$ tiene una complejidad de $O(n)$ en el peor caso.
- La función $prefix - reversal$ tiene una complejidad de $O(n)$ ya que involucra la inversión de una sublista de tamaño n.
- En el peor caso, el bucle interno se ejecuta n veces, lo que da una complejidad de $O(n^2)$ en el peor caso.

Por lo tanto, la complejidad computacional total del algoritmo es de $O(n^2)$ en el peor caso, donde n es el tamaño de la permutación. Esto se debe a que el bucle interno anidado puede ejecutarse hasta n veces en el peor escenario, lo que contribuye a la complejidad cuadrática del algoritmo (ver figura 42).

Figura 42

Análisis asintótico de soncco2013 vs. al2014 vs. soncco2014 vs. propuesta



Nota. Complejidad computacional ***soncco2013*** *vs.* ***al2014*** *vs.* ***soncco2014*** *vs.* ***propuesta***

Como consecuencia de los resultados obtenidos, las siguientes tablas (tabla 3 y 4) muestran un resumen de los mejores algoritmos según su complejidad computacional.

Por otro lado, la tabla 5 muestra que el mejor algoritmo metaheurístico ha sido soncco2014 y en contraposición el mejor algoritmo heurístico ha sido el propuesto denominado reversión de prefijo de doble lado, que por simplificación, ambos poseen una complejidad computacional lineal.

Esto implica que al emplear estos algoritmos para analizar el comportamiento del problema SBPR, se obtendrán resultados de manera más rápida en comparación con cualquier otro algoritmo propuesto o analizado en este estudio. Es importante destacar que esta mejora en la velocidad no resuelve el problema por completo, dado que no es factible prever con certeza la salida exacta de $f(n)$, a partir de una entrada

n. Sin embargo, la aplicación de estos métodos abre nuevas posibilidades para facilitar la comprensión del problema en el futuro, lo que podría acelerar el proceso hacia una solución definitiva en un plazo más breve, y así colaborar en los diferentes avances de esta línea de investigación.

Tabla 3

Resultados de algoritmos metaheurísticos

Autor y año	**Complejidad computacional**
Soncco-Álvarez y Ayala-Rincón (2013)	$O(n^2 log10(n))$
Al Daoud (2014)	$O(n^2)$
Soncco-Álvarez y Ayala-Rincón (2014)	$O(2n)$
Algoritmo metaheurístico propuesto	$O(n^2)$

Nota. El algoritmo metaheurístico soncco2014 es el más eficiente de todos los analizados

Tabla 4

Resultados de algoritmos heurísticos

Autor y año	**Complejidad computacional**
U. Dias et al. (2014)	$O(n!)$
Brito et al. (2019)	$O(n^2 + n)$
Algoritmo heurístico propuesto	$O(n)$

Nota. El algoritmo heurístico propuesto es el más eficiente de todos los analizados

Tabla 5

Algoritmos con mejores resultados en complejidad computacional

Autor y año	**Complejidad computacional**	**Tipo de algoritmo**
Soncco-Álvarez y Ayala-Rincón (2014)	$O(2n)$	Metaheurístico
Algoritmo heurístico propuesto	$O(n)$	Heurístico

Nota. Tanto el algoritmo metaheurístico soncco2014 como el algoritmo heurístico propuesto son los más eficientes de todos los analizados.

Conclusiones

7. CONCLUSIONES

En esta tesis se ha trazado un recorrido cronológico del problema SBPR, aplicado al ámbito de la genética computacional. Se ha examinado cada hito significativo en su evolución histórica, y se ha destacado cómo los diversos resultados obtenidos han dado lugar a nuevas áreas de investigación que han contribuido a abordar problemas reales. Ante esto se presentan las siguientes conclusiones:

- Se presentó una descripción teórica exhaustiva de las disciplinas matemáticas fundamentales para desarrollar soluciones exactas, heurísticas y metaheurísticas al problema SBPR. Se examinaron en detalle la teoría combinatoria clásica, el álgebra lineal, la complejidad computacional y el análisis asintótico, proporcionando el marco teórico necesario para el desarrollo de los algoritmos propuestos en esta tesis.

- Se detalló la estructura de los algoritmos mediante el análisis de su pseudocódigo. Este análisis fue crucial para la implementación de los procedimientos en código, permitiendo replicar los resultados obtenidos en cada estudio.

- Los cinco algoritmos previamente analizados, junto con dos nuevas propuestas de mejora, se implementaron en Python utilizando la plataforma en línea Google Colab. Esta elección proporcionó la flexibilidad necesaria para realizar todas las pruebas de rendimiento de los algoritmos. Adicionalmente, en algunos casos descritos en el capítulo 5, se utilizó una computadora personal con hardware de alta calidad para verificar si los tiempos de ejecución observados en Google Colab eran consistentes con los obtenidos localmente.

- Finalmente, según los resultados obtenidos, el algoritmo heurístico más destacado fue el de reversión de prefijo de doble lado, con una complejidad computacional de $O(n)$. En cuanto a los algoritmos metaheurísticos, nuestra propuesta inspirada en los trabajos de Bafna y Pevzner mostró una complejidad computacional solo un orden inferior a la de soncoo2014 que es de $O(2n)$.

Recomendaciones

Basado en las conclusiones anteriores y las nuevas oportunidades que se abren en esta línea de investigación, se puede afirmar lo siguiente:

- Según el principio de reproducibilidad de la ciencia, este trabajo contiene una forma directa de verificar sus resultados utilizando el código Python generado para cada uno de los algoritmos desde el repositorio público de Github[18].
- Se recomienda continuar con las siguientes tres vías para ampliar esta línea de investigación:
    - Buscar mejoras en los algoritmos propuestos, siguiendo la recomendación anterior.
    - Desarrollar un análisis comparativo basado en los estadísticos propuestos por Du y Swamy (2016) y autores relacionados.
    - Desarrollar propuestas de algoritmos de paralelización como los que propone Soncco-Álvarez et al. (2013) e implementar una comparativa basada en análisis asintótico o utilizando los métodos estadísticos de Du y Swamy (2016).
    - Clasificar el catálogo de algoritmos metaheurísticos propuestos por Yang y He (2019) y Yang (2020), esto permitirá implementar los procedimientos que se adecuen al problema en primer lugar y finalmente compararlos utilizando las dos técnicas mencionadas en el punto anterior.

Perspectiva de la investigación

Como trabajo futuro, se terminarán los algoritmos expuestos y se brindará un análisis de sus resultados más detallado. Tomando en cuenta los parámetros planteados por Du y Swamy (2016) y los que utilizó cada autor de forma individual.

Posteriormente, se planteará un algoritmo metaheurístico que mejore las resultados existentes de **soncoo2014** utilizando siempre la misma metodología de

[18]Es importante mencionar que este repositorio puede sufrir modificaciones en el tiempo si se encuentran nuevas soluciones o procedimientos más óptimos que mejoren los actuales. La idea de un repositorio público, es que diferentes desarrolladores, investigadores, matemáticos o aficionados puedan contribuir a realizar mejoras sobre este planteamiento inicial.

análisis. A la vez, se podría plantear un nuevo algoritmo heurístico que mejore los resultados existentes de **reversión de prefijo de doble lado**.

En caso de que las soluciones no sean lo suficientemente robustas se buscarán enfoques alternativos como algoritmos de paralelización Soncco-Álvarez et al. (2013) o una revisión más completa del enorme catálogo de metaheurísticas basadas en la vida, basadas en el trabajo de Yang y He (2019) y Yang (2020).

Referencias

REFERENCIAS

Al Daoud, E. (2014). An Improved Ant Colony Algorithm for Genome Rearrangements. *International Journal of Bioengineering and Life Sciences*, *8*(5), 768-771.

Baase, S., Van Gelder, A., & García, R. L. E. (2002). *Algoritmos computacionales: Introducción al análisis y diseño*. Pearson Educación.

Bafna, V., & Pevzner, P. A. (1996). Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, *25*(2), 272-289.

Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, *2*(4), 353-373.

Bóna, M. (2012). *Combinatorics of permutations*. CRC Press.

Brito, K. L., Oliveira, A. R., Dias, U., & Dias, Z. (2019). Heuristics for the reversal and transposition distance problem. *IEEE/ACM transactions on computational biology and bioinformatics*, *17*(1), 2-13.

Bulteau, L. (2013). *Algorithmic aspects of genome rearrangements* [Tesis doctoral, Université de Nantes].

Bulteau, L., Fertin, G., & Rusu, I. (2012). Pancake flipping is hard. *Proceedings of the 37th international conference on Mathematical Foundations of Computer Science*, 247-258.

Caprara, A., Lancia, G., & Ng, S.-K. (1999). *A column-generation based branch-and-bound algorithm for sorting by reversals* (Vol. 47).

Chitturi, B., Fahle, W., Meng, Z., Morales, L., Shields, C. O., Sudborough, I. H., & Voit, W. (2009). An (18/11) n upper bound for sorting by prefix reversals. *Theoretical Computer Science*, *410*(36), 3372-3390.

Christie, D. A. (1998). A 3/2-approximation algorithm for sorting by reversals. *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, 244-252.

da Silveira, L. A., Soncco-Alvarez, J. L., de Lima, T. A., & Ayala-Rincon, M. (2018). Parallel multi-island genetic algotirth for sorting unsigned genomes by reversals. *2018 IEEE Congress on Evolutionary Computation (CEC)*, 1-8.

da Silveira, L. Â., Soncco-Álvarez, J. L., & Ayala-Rincón, M. (2017). Parallel genetic algorithms with sharing of individuals for sorting unsigned genomes by reversals. *2017 IEEE Congress on Evolutionary Computation (CEC)*, 741-748.

de Cabezón Irigaray, E. S. (2020). *£Sabes qué son los NÚMEROS TORTITAS o NÚMEROS PANCAKE?* Youtube. https://www.youtube.com/watch?v=uVnxqEROYmA&t=249s&ab_channel=Derivando

Dias, U., Galvão, G. R., Lintzmayer, C. N., & Dias, Z. (2014). A general heuristic for genome rearrangement problems. *Journal of bioinformatics and computational biology*, *12*(03), 1450012.

Dias, Z., & Dias, U. (2015). Sorting by prefix reversals and prefix transpositions. *Discrete Applied Mathematics*, *181*, 78-89.

Dorigo, M., & Blum, C. (2005). Ant colony optimization theory: A survey. *Theoretical computer science*, *344*(2-3), 243-278.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *26*(1), 29-41.

Du, K.-L., & Swamy, M. (2016). *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature*. Birkhäuser.

Dweighter, H. (1975). American Mathematical Monthly. *The Mathematical Association of America*, *150*, 1010.

Dweighter, H., Garey, M. R., Johnson, D. S., & Lin, S. (1977). E2569. *The American Mathematical Monthly*, *84*(4), 296-296.

Fertin, G., Labarre, A., Rusu, I., Vialette, S., & Tannier, E. (2009). *Combinatorics of genome rearrangements*. MIT press.

Fischer, J., & Ginzinger, S. W. (2005). A 2-approximation algorithm for sorting by prefix reversals. *European Symposium on Algorithms*, 415-425.

Gates, W. H., & Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete mathematics*, *27*(1), 47-57.

Haines, J. (2017, septiembre). *The Simplicity of Computational Complexity: Street Fighter II VS. the Big O | HackerNoon*. https://hackernoon.com/the-simplicity-of-computational-complexity-street-fighter-ii-vs-the-big-o-67786db7653c

Helmert, M. (2010). Landmark heuristics for the pancake problem. *Third Annual Symposium on Combinatorial Search*.

Hillier, F. S., & Lieberman, G. J. (2010). *Introducción a la investigación de operaciones*. McGraw Hill Educación.

Kramer, O. (2017). Genetic algorithms. En *Genetic algorithm essentials* (pp. 11-19). Springer.

Lintzmayer, C. N. (2016). *The Problem of Sorting Permutations by Prefix and Suffix Rearrangements* [Tesis doctoral, PhD thesis, University of Campinas, Institute of Computing, 2016. In English].

López, W. J. P., & Gómez, F. J. H. (2023). Incidencia de los estudios sobre reordenamiento genómico en la secuenciación del genoma humano. *Revista Científica Estelí*, (46), 262-278.

Meijer, A. (2020). *Bill Gates wird Mitbesitzer des Hotel Formentor auf Mallorca*. https://www.mallorcamagazin.com/nachrichten/lokales/2020/05/27/80655/bill-gates-wird-aktionar-hotel-formentor-auf-mallorca.html

Moscato, P., & Cotta, C. (2003). Una introducción a los algoritmos meméticos. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, *7*(19).

Nurk, S., Koren, S., Rhie, A., Rautiainen, M., Bzikadze, A. V., Mikheenko, A., Vollger, M. R., Altemose, N., Uralsky, L., Gershman, A., Aganezov, S., Hoyt, S. J., Diekhans, M., Logsdon, G. A., Alonge, M., Antonarakis, S. E., Borchers, M.,

Bouffard, G. G., Brooks, S. Y., ... Phillippy, A. M. (2022). The complete sequence of a human genome. *Science*, *376*(6588), 44-53. https://doi.org/10.1126/science.abj6987

Okwu, M. O., & Tartibu, L. K. (2020). *Metaheuristic optimization: Nature-inspired algorithms swarm and computational intelligence, theory and applications* (Vol. 927). Springer Nature.

Pelta Mochcovsky, D. A. (2013). Algoritmos heurísticos en bioinformática. *Tesis Univ. Granada. Departamento de la Computación e Inteligencia Artificial*. http://hdl.handle.net/10481/24513

Rudolph, G. (1994). Convergence analysis of canonical genetic algorithms. *IEEE transactions on neural networks*, *5*(1), 96-101.

Sharmin, M., Yeasmin, R., & Hasan, M. (2008). Sorting by Prefix Reversals and Prefix Transpositions with Forward March. *arXiv preprint arXiv:0812.3933*.

Sheppard, C. (2017). *Genetic algorithms with Python*. Smashwords Edition S. l.

Sjøberg, D. I. (2010). Confronting the myth of rapid obsolescence in computing research. *Communications of the ACM*, *53*(9), 62-67.

Soncco-Álvarez, J. L., Almeida, G. M., Becker, J., & Ayala-Rincon, M. (2013). Parallelization and virtualization of genetic algorithms for sorting permutations by reversals. *2013 World Congress on Nature and Biologically Inspired Computing*, 29-35.

Soncco-Álvarez, J. L., & Ayala-Rincón, M. (2012). A genetic approach with a simple fitness function for sorting unsigned permutations by reversals. *2012 7th Colombian Computing Congress (CCC)*, 1-6.

Soncco-Álvarez, J. L., & Ayala-Rincón, M. (2013). Sorting permutations by reversals through a hybrid genetic algorithm based on breakpoint elimination and exact solutions for signed permutations. *Electronic Notes in Theoretical Computer Science*, *292*, 119-133.

Soncco-Álvarez, J. L., & Ayala-Rincón, M. (2014). Memetic algorithm for sorting unsigned permutations by reversals. *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2770-2777.

Sryheni, S. (2020). Sliding Window Algorithm | Baeldung on Computer Science [Accessed on 06/08/2022].

Yang, X.-S. (2020). *Nature-inspired optimization algorithms*. Academic Press.

Yang, X.-S., & He, X.-S. (2019). *Mathematical foundations of nature-inspired algorithms*. Springer.

Zhongxi, M., & Tao, Z. (2006). An improved genetic algorithm for problem of genome rearrangement. *Wuhan University Journal of Natural Sciences*, *11*(3), 498-502.

Anexos

ANEXOS

Heurísticas

Implementacion del algoritmo **dias2014:**

```
1
2   # Define the reduce function
3   def prefix_reversal(sequence, start, end):
4       return sequence[:start] + sequence[start:end][::-1] + sequence[
    end:]
5
6   def reduce(sequence):
7       reduced_sequence = sequence
8       for i in range(len(sequence)):
9           for j in range(i+1, len(sequence)+1):
10              reversed_sequence = prefix_reversal(sequence, i, j)
11              if len(reversed_sequence) < len(reduced_sequence):
12                  reduced_sequence = reversed_sequence
13      return reduced_sequence
14
15  # Define the database
16
17  """
18  database = {
19      '123': '123',  # Example: permutation '123' maps to itself
20      '132': '123',  # Example: permutation '132' maps to '123'
21      # Add more entries for other permutations and their optimal
```

```
solutions
}
"""
database = {"123": "123", "132": "123", "213": "123", "231": "132",
"312": "132", "321": "123"}

# Define Algorithm 1: Sliding Window
def sliding_window(sequence):
    start = 0
    end = 1
    best_end = 1
    best_size = len(sequence)
    while end <= len(sequence):
        window = sequence[start:end]
        reduced_window = reduce(window)
        if len(reduced_window) <= 12:
            size = len(reduced_window)
            if size < best_size:
                best_size = size
                best_end = end
        if end == len(sequence):
            break
        end += 1
        if end - start > 12:
            start += 1
```

```
45          return sequence[:best_end]
46
47      # Define Algorithm 2: Database Lookup
48      def database_lookup(sequence, database):
49          best_sequence = sequence
50          for i in range(len(sequence)):
51              for j in range(i+1, len(sequence)+1):
52                  subsequence = sequence[i:j]
53                  if len(subsequence) <= 12:
54                      if subsequence in database:
55                          if len(database[subsequence]) < len(
    best_sequence):
56                              best_sequence = database[subsequence]
57          return best_sequence
58
59      # Example usage
60      sequence = '312'
61      reduced_sequence = reduce(sequence)
62      print(f"Reduced sequence: {reduced_sequence}")
63
64      sliding_window_sequence = sliding_window(sequence)
65      print(f"Sliding window sequence: {sliding_window_sequence}")
66
67      database_sequence = database_lookup(sequence, database)
68      print(f"Database sequence: {database_sequence}")
```

Implementación del algoritmo propuesto para generar base de datos clave valor en referencia a **dias2014:**

```python

import json
from itertools import permutations

def prefix_reversals_database(n):
    """
    Generates a database of optimal solutions for the prefix reversals problem
    for all permutations of length n.
    """
    database = {}
    for perm in permutations(range(1, n+1)):
        perm_str = ''.join(str(x) for x in perm)
        database[perm_str] = perm_str
        for i in range(1, n):
            for j in range(i+1, n+1):
                rev_perm = perm[:i-1] + perm[i-1:j][::-1] + perm[j:]
                rev_perm_str = ''.join(str(x) for x in rev_perm)
                if rev_perm_str < database[perm_str]:
                    database[perm_str] = rev_perm_str
    return database


n = 9
```

```
24    database = prefix_reversals_database(n)
25    with open(f"data-{n}.txt", "w") as data_file:
26        data_file.write(json.dumps(database))
27    print(database)
```

Implementación del algoritmo **brito2019:**

```
1
2   def distance_estimate(arr1, arr2):
3       # Calcula la distancia entre dos permutaciones contando los
    elementos diferentes
4       return sum(1 for x, y in zip(arr1, arr2) if x != y)
5
6   def look_ahead(permutation, target_permutation):
7       current_permutation = permutation.copy()
8       while current_permutation != target_permutation:
9           best_permutation = None
10          best_distance = float('inf')
11          # Explora el vecindario de la permutacion actual
12          for i in range(len(current_permutation)):
13              for j in range(i+1, len(current_permutation)):
14                  new_permutation = current_permutation[:i] +
    current_permutation[i:j+1][::-1] + current_permutation[j+1:]
15                  # Estima la distancia entre la nueva permutacion y
    la permutacion objetivo
16                  distance = distance_estimate(new_permutation,
    target_permutation)
17                  if distance < best_distance:
18                      best_distance = distance
19                      best_permutation = new_permutation
20          if best_permutation is None:
```

```
21                 break  # Salir del bucle si no se encuentra una mejor
    permutacion
22             current_permutation = best_permutation
23         return current_permutation
24
25     # Ejemplo de uso para el problema Sorting Permutations by Prefix
    Reversals
26     permutation = [3, 1, 2, 4]
27     target_permutation = [1, 2, 3, 4]
28     sorted_permutation = look_ahead(permutation, target_permutation)
29     print("Permutacion ordenada:", sorted_permutation)
```

Metaheurísticas

Implementación del algoritmo **soncco2013:**

```
1
2   import random
3   import statistics
4
5   fitness_scores_list = []
6
7   # Function to generate a random unsigned permutation of a given
    length
8   def generate_random_permutation(length):
9       return random.sample(range(1, length+1), length)
10
11  def calculate_reversal_distance(permutation):
12      n = len(permutation)
13      visited = [False] * n
14      reversal_distance = 0
15
16      for i in range(n):
17          if not visited[i]:
18              j = i
19              cycle_length = 0
20              while not visited[j]:
21                  visited[j] = True
22                  j = permutation[j] - 1
```

```
                    cycle_length += 1
                if cycle_length > 1:
                    reversal_distance += cycle_length - 1

        return reversal_distance

# Function to perform tournament selection on a population of candidate solutions
def tournament_selection(population, fitness_scores, tournament_size=2):
        selected_parents = []
        for i in range(len(population)):
            tournament = random.sample(range(len(population)), tournament_size)
            winner = tournament[0]
            for j in range(1, tournament_size):
                if fitness_scores[tournament[j]] < fitness_scores[winner]:
                    winner = tournament[j]
            selected_parents.append(population[winner])
        return selected_parents

# Function to perform single-point crossover on two parent solutions
def perform_crossover(parent1, parent2):
        length = len(parent1)
```

```
44        crossover_point = random.randint(1, length-1)
45        child1 = parent1[:crossover_point] + parent2[crossover_point:]
46        child2 = parent2[:crossover_point] + parent1[crossover_point:]
47        return child1, child2
48
49    # Function to perform mutation on a candidate solution
50    def perform_mutation(solution):
51        length = len(solution)
52        mutation_point1 = random.randint(0, length-1)
53        mutation_point2 = random.randint(0, length-1)
54        while mutation_point2 == mutation_point1:
55            mutation_point2 = random.randint(0, length-1)
56        if mutation_point1 > mutation_point2:
57            mutation_point1, mutation_point2 = mutation_point2,
    mutation_point1
58        mutated_solution = solution[:mutation_point1] + list(reversed(
    solution[mutation_point1:mutation_point2+1])) + solution[
    mutation_point2+1:]
59        return mutated_solution
60
61    # Function to select the best candidate solutions from a combined
    population of parents and offspring
62    def select_best_candidates(combined_population, fitness_scores,
    num_candidates):
63        sorted_population = [x for _, x in sorted(zip(fitness_scores,
```

```
combined_population))]
    return sorted_population[:num_candidates]

# Function to select the best candidate solution from a population of candidate solutions
def select_best_solution(population, fitness_scores):
    best_index = fitness_scores.index(min(fitness_scores))
    return population[best_index]

# Genetic algorithm for sorting by reversals of unsigned permutations
def genetic_algorithm_for_unsigned_permutations(initial_permutation, population_size, num_generations):
    # Generate initial population of candidate solutions
    population = [generate_random_permutation(len(initial_permutation)) for _ in range(population_size)]

    for generation in range(num_generations):
        # Evaluate fitness of each candidate solution
        fitness_scores = [calculate_reversal_distance(perm) for perm in population]

        #print(f"- reversal distance: {calculate_reversal_distance(perm)}")

```

```
        # Selection
        selected_parents = tournament_selection(population, fitness_scores)

        # Crossover
        offspring = []
        for i in range(0, len(selected_parents), 2):
            parent1 = selected_parents[i]
            parent2 = selected_parents[i+1]
            child1, child2 = perform_crossover(parent1, parent2)
            offspring.extend([child1, child2])

        # Mutation
        mutated_offspring = [perform_mutation(perm) for perm in offspring]

        # Evaluate fitness of offspring
        offspring_fitness_scores = [calculate_reversal_distance(perm) for perm in mutated_offspring]

        # Replacement
        combined_population = population + mutated_offspring
        population = select_best_candidates(combined_population, offspring_fitness_scores, population_size)

```

```
        # Return the best candidate solution found
        best_solution = select_best_solution(population, fitness_scores)
        return best_solution

    def quicksort(array):
        if len(array) <= 1:
            return array
        else:
            pivot = array[0]
            less = [x for x in array[1:] if x <= pivot]
            greater = [x for x in array[1:] if x > pivot]
            return quicksort(less) + [pivot] + quicksort(greater)


    final_list = []
    # iterations
    for i in range(2, 101):
        print(f"Size of permutation: {i}")
        population_size = 50
        num_generations = 30
        random_permutation = generate_random_permutation(i*10)
        print(random_permutation)
        best_solution = genetic_algorithm_for_unsigned_permutations(
random_permutation, population_size, num_generations)
        final_list.append(calculate_reversal_distance(best_solution))
```

```
127         print(f"Best solution: {best_solution} - \nSorted: {quicksort(
    best_solution)}")
128         print("**********************\n")
```

Implementación del algoritmo **al2014:**

```
import random

def breakpoint_reversal_sort(permutation):
    def find_breakpoints(perm):
        breakpoints = 0
        for i in range(len(perm) - 1):
            if abs(perm[i] - perm[i + 1]) != 1:
                breakpoints += 1
        if perm[0] != 1:
            breakpoints += 1
        if perm[-1] != len(perm):
            breakpoints += 1
        return breakpoints

    def reverse_strip(perm, start, end):
        return perm[:start] + list(reversed(perm[start:end + 1])) +
perm[end + 1:]

    def find_decreasing_strip(perm):
        for i in range(len(perm) - 1):
            if perm[i] > perm[i + 1]:
                j = i
                while j < len(perm) - 1 and perm[j] > perm[j + 1]:
                    j += 1
```

```
24                      return i, j
25              return None
26
27          def find_increasing_strip(perm):
28              for i in range(len(perm) - 1):
29                  if perm[i] < perm[i + 1]:
30                      j = i
31                      while j < len(perm) - 1 and perm[j] < perm[j + 1]:
32                          j += 1
33                      return i, j
34              return None
35
36          def perform_reversal(perm, start, end):
37              if start == 0:
38                  return list(reversed(perm[:end + 1])) + perm[end + 1:]
39              else:
40                  return perm[:start] + list(reversed(perm[start:end + 1])
    ) + perm[end + 1:]
41
42          # Main algorithm
43          while find_breakpoints(permutation) > 0:
44              decreasing_strip = find_decreasing_strip(permutation)
45              if decreasing_strip:
46                  permutation = perform_reversal(permutation,
    decreasing_strip[0], decreasing_strip[1])
```

```
            else:
                increasing_strip = find_increasing_strip(permutation)
                permutation = perform_reversal(permutation, increasing_strip[0], increasing_strip[1])

        return permutation


perm = random.sample(range(1, 21), 20)

for i in range(len(perm) - 1):
    if perm[i] > perm[i + 1]:
        break
else:
    print("Permutation does not meet criteria of unsigned prefix reversals problem.")

permu = breakpoint_reversal_sort(perm)
print(f"unsorted: {perm}")
print(f"sorted: {permu}")
```

Implementación del algoritmo **soncco2014:**

```
import random

# Funcion para calcular la distancia de reversion de una permutacion
def reversal_distance(permutation, sorted_permutation):
    distance = 0
    for i in range(len(permutation)):
        if permutation[i] != sorted_permutation[i]:
            distance += 1
    return distance

# Funcion para realizar una inversion de prefijo en una permutacion
def prefix_reversal(permutation, idx):
    return permutation[:idx][::-1] + permutation[idx:]

# Algoritmo memetico para ordenar una permutacion por inversiones de prefijo
def memetic_algorithm(permutation, max_iterations, population_size):
    n = len(permutation)
    sorted_permutation = list(range(1, n+1))

    # Inicializar poblacion con permutaciones aleatorias
    population = [random.sample(range(1, n+1), n) for _ in range(population_size)]

```

```
23     for _ in range(max_iterations):
24         # Aplicar inversion de prefijo a cada individuo de la poblacion
25         for i in range(population_size):
26             idx = random.randint(0, n-1)
27             population[i] = prefix_reversal(population[i], idx)
28
29         # Seleccionar el mejor individuo de la poblacion
30         best_permutation = min(population, key=lambda x:
    reversal_distance(x, sorted_permutation))
31
32         # Verificar si se ha alcanzado la permutacion ordenada
33         if best_permutation == sorted_permutation:
34             return best_permutation
35
36     return best_permutation
37
38 # Ejemplo de uso
39 permutation = [6,3,1,4,5,2,9,7,8]
40 max_iterations = 1000
41 population_size = 500
42 result = memetic_algorithm(permutation, max_iterations, population_size)
43
44 print("Permutacion original:", permutation)
45 print("Permutacion ordenada:", result)
```

Algoritmos propuestos

Implementación del algoritmo de “reversión“ de doble lado (heurístico):

```
1  def two_sided_prefix_reversal_sort(permutation):
2  sorted_permutation = sorted(permutation)
3  operations = []  # Lista para almacenar las operaciones de reversion
   de prefijo
4
5  left, right = 0, len(permutation) - 1
6
7  while left < right:
8      if permutation[left] == sorted_permutation[left]:
9          left += 1
10     elif permutation[right] == sorted_permutation[right]:
11         right -= 1
12     else:
13         if left < right:
14             permutation[left:right+1] = permutation[left:right
   +1][::-1]
15             operations.append((left, right))
16         left += 1
17         right -= 1
18
19 # Verificar si la permutacion está ordenada y corregir si es
   necesario
20 if permutation != sorted_permutation:
```

```
        # Aplicar una operacion final para ordenar la permutacion
        permutation = sorted_permutation
        operations.append((0, len(permutation) - 1))

    return permutation, operations

# Ejemplo de uso
permutation = [3, 1, 2, 4, 6, 5, 8,7,10,9,11,14,13,12]
sorted_permutation, operations = two_sided_prefix_reversal_sort(
    permutation)
print("Permutacion ordenada:", sorted_permutation)
print("Operaciones de reversion de prefijo:", operations)
```

Implementación del algoritmo Bafna y Pevzner (metaheurístico):

```
import random

def prefix_reversal_sort_bafna_pevzner(permutation, max_iterations, restart_threshold):
    n = len(permutation)
    sorted_permutation = list(permutation)
    iterations_without_improvement = 0

    # Funcion para realizar una inversion de prefijo en la permutacion
    def prefix_reversal(p, i, j):
        return p[:i] + p[i:j+1][::-1] + p[j+1:]

    for _ in range(max_iterations):
        if iterations_without_improvement >= restart_threshold:
            sorted_permutation = list(range(1, n+1))  # Reiniciar con una permutacion aleatoria
            iterations_without_improvement = 0

        improved = False
        for i in range(n):
            if sorted_permutation[i] != i+1:
                j = sorted_permutation.index(i+1)
                new_permutation = prefix_reversal(sorted_permutation
```

```
, i, j)
                    if new_permutation != sorted_permutation:
                        sorted_permutation = new_permutation
                        improved = True
                        iterations_without_improvement = 0
                        break

            if not improved:
                iterations_without_improvement += 1

        return sorted_permutation

# Ejemplo de uso
# Busqueda local con reinicio aleatorio
max_iterations = 1000
restart_threshold = 50
initial_permutation = [4, 3, 2, 1, 7, 6, 5]
sorted_permutation = prefix_reversal_sort_bafna_pevzner(
initial_permutation, max_iterations, restart_threshold)
print("Permutacion Inicial:", initial_permutation)
print("Permutacion Ordenada:", sorted_permutation)
```

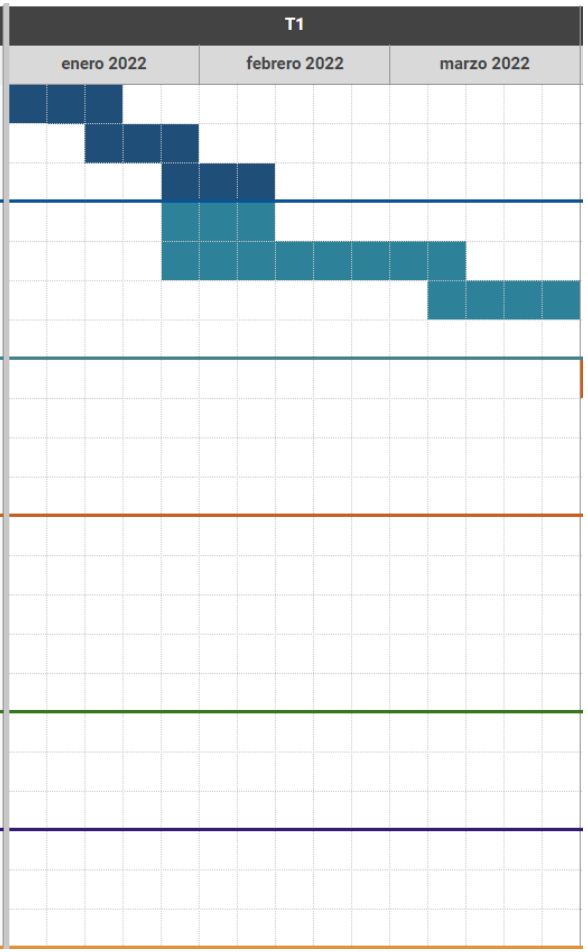
Repositorios de Github

- Enlace público de algoritmos en Github
- Enlace del código látex de este documento de tesis

Cronograma de la investigación

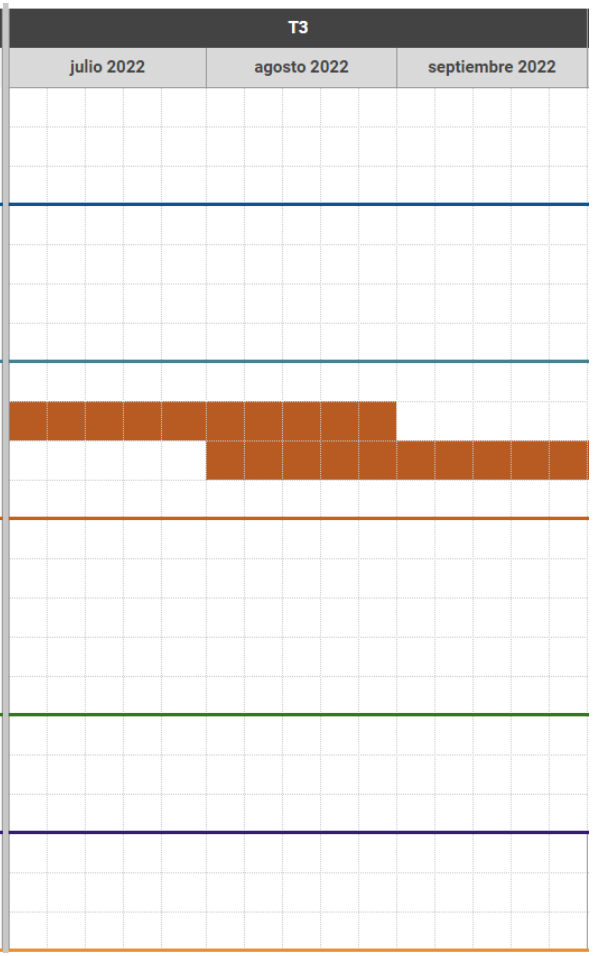
A continuación se muestra el calendario de actividades desarrollado a lo largo de este programa de posgrado, incluyendo las actividades de elaboración de tesis.

FASE		DETALLES
1	**Idea de tesis**	Propuesta de idea a coordinacion Propuesta de idea al tutor Elaboración del bosquejo inicial
2	**Publicaciones de maestría**	Inscripcion en evento de XXXIII SIMMAC 2022 UCR Elaboración de resumen para XXXIII SIMMAC 2022 Participacion en evento XXXIII SIMMAC 2022
3	**Desarrollo del protocolo tesis maestría**	Estado del arte Marco teórico Resultados Defensa de protocolo
4	**Diseño y Programación de los Algoritmos de la tesis doctoral**	Implementación de los Algoritmos Heurísticos Implementación de los algoritmos Metaheuristicos Implementación de propuesta heurística Implementación de propuesta metaheurística Diseño y orden de complejodad de los algoritmos propuestos
5	**Publicaciones de doctorado**	Conferencia Keiser University Papers para publicación nacional - primera ronda Papers para publicación nacional - segunda ronda
6	**Desarrollo de tesis doctoral**	Mejoras en el marco teórico Mejoras en los resultados Predefensa y defensa



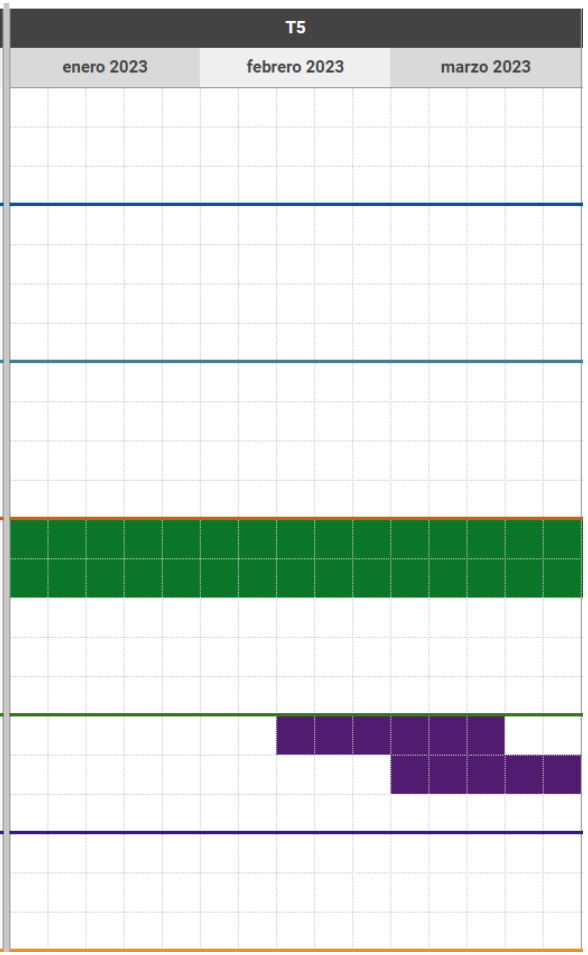
FASE		DETALLES	T2		
			abril 2022	mayo 2022	junio 2022
1	Idea de tesis	Propuesta de idea a coordinacion			
		Propuesta de idea al tutor			
		Elaboración del bosquejo inicial			
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR			
		Elaboración de resumen para XXXIII SIMMAC 2022			
		Participacion en evento XXXIII SIMMAC 2022			
3	Desarrollo del protocolo tesis maestría	Estado del arte			
		Marco teórico			
		Resultados			
		Defensa de protocolo			
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos			
		Implementación de los algoritmos Metaheuristicos			
		Implementación de propuesta heurística			
		Implementación de propuesta metaheurística			
		Diseño y orden de complejodad de los algoritmos propuestos			
5	Publicaciones de doctorado	Conferencia Keiser University			
		Papers para publicación nacional - primera ronda			
		Papers para publicación nacional - segunda ronda			
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico			
		Mejoras en los resultados			
		Predefensa y defensa			

FASE		DETALLES
1	Idea de tesis	Propuesta de idea a coordinacion
		Propuesta de idea al tutor
		Elaboración del bosquejo inicial
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR
		Elaboración de resumen para XXXIII SIMMAC 2022
		Participacion en evento XXXIII SIMMAC 2022
3	Desarrollo del protocolo tesis maestría	Estado del arte
		Marco teórico
		Resultados
		Defensa de protocolo
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos
		Implementación de los algoritmos Metaheuristicos
		Implementación de propuesta heurística
		Implementación de propuesta metaheurística
		Diseño y orden de complejodad de los algoritmos propuestos
5	Publicaciones de doctorado	Conferencia Keiser University
		Papers para publicación nacional - primera ronda
		Papers para publicación nacional - segunda ronda
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico
		Mejoras en los resultados
		Predefensa y defensa



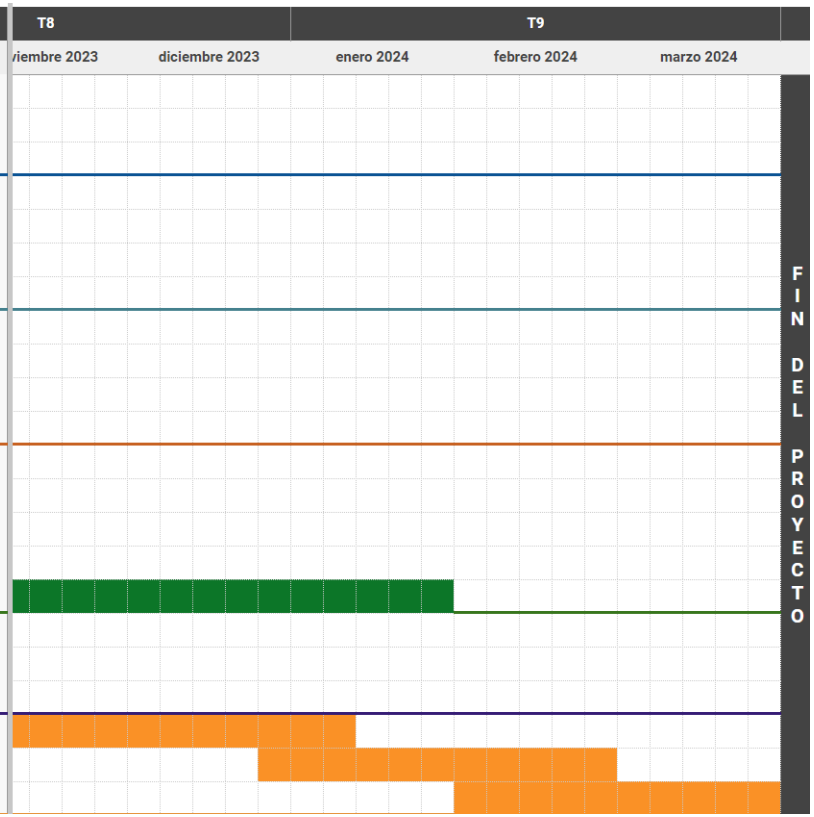
FASE		DETALLES	T4		
			octubre 2022	noviembre 2022	diciembre 2022
1	**Idea de tesis**	Propuesta de idea a coordinacion			
		Propuesta de idea al tutor			
		Elaboración del bosquejo inicial			
2	**Publicaciones de maestría**	Inscripcion en evento de XXXIII SIMMAC 2022 UCR			
		Elaboración de resumen para XXXIII SIMMAC 2022			
		Participacion en evento XXXIII SIMMAC 2022			
3	**Desarrollo del protocolo tesis maestría**	Estado del arte			
		Marco teórico			
		Resultados			
		Defensa de protocolo			
4	**Diseño y Programación de los Algoritmos de la tesis doctoral**	Implementación de los Algoritmos Heurísticos			
		Implementación de los algoritmos Metaheuristicos			
		Implementación de propuesta heurística			
		Implementación de propuesta metaheurística			
		Diseño y orden de complejodad de los algoritmos propuestos			
5	**Publicaciones de doctorado**	Conferencia Keiser University			
		Papers para publicación nacional - primera ronda			
		Papers para publicación nacional - segunda ronda			
6	**Desarrollo de tesis doctoral**	Mejoras en el marco teórico			
		Mejoras en los resultados			
		Predefensa y defensa			

FASE		DETALLES
1	Idea de tesis	Propuesta de idea a coordinacion Propuesta de idea al tutor Elaboración del bosquejo inicial
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR Elaboración de resumen para XXXIII SIMMAC 2022 Participacion en evento XXXIII SIMMAC 2022
3	Desarrollo del protocolo tesis maestría	Estado del arte Marco teórico Resultados Defensa de protocolo
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos Implementación de los algoritmos Metaheuristicos Implementación de propuesta heurística Implementación de propuesta metaheurística Diseño y orden de complejodad de los algoritmos propuestos
5	Publicaciones de doctorado	Conferencia Keiser University Papers para publicación nacional - primera ronda Papers para publicación nacional - segunda ronda
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico Mejoras en los resultados Predefensa y defensa



FASE		DETALLES	T6		
			abril 2023	mayo 2023	junio 2023
1	Idea de tesis	Propuesta de idea a coordinacion			
		Propuesta de idea al tutor			
		Elaboración del bosquejo inicial			
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR			
		Elaboración de resumen para XXXIII SIMMAC 2022			
		Participacion en evento XXXIII SIMMAC 2022			
3	Desarrollo del protocolo tesis maestría	Estado del arte			
		Marco teórico			
		Resultados			
		Defensa de protocolo			
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos			
		Implementación de los algoritmos Metaheuristicos			
		Implementación de propuesta heurística			
		Implementación de propuesta metaheurística			
		Diseño y orden de complejodad de los algoritmos propuestos			
5	Publicaciones de doctorado	Conferencia Keiser University			
		Papers para publicación nacional - primera ronda			
		Papers para publicación nacional - segunda ronda			
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico			
		Mejoras en los resultados			
		Predefensa y defensa			

FASE		DETALLES	T7		
			julio 2023	agosto 2023	septiembre 2023
1	**Idea de tesis**	Propuesta de idea a coordinacion			
		Propuesta de idea al tutor			
		Elaboración del bosquejo inicial			
2	**Publicaciones de maestría**	Inscripcion en evento de XXXIII SIMMAC 2022 UCR			
		Elaboración de resumen para XXXIII SIMMAC 2022			
		Participacion en evento XXXIII SIMMAC 2022			
3	**Desarrollo del protocolo tesis maestría**	Estado del arte			
		Marco teórico			
		Resultados			
		Defensa de protocolo			
4	**Diseño y Programación de los Algoritmos de la tesis doctoral**	Implementación de los Algoritmos Heurísticos			
		Implementación de los algoritmos Metaheuristicos			
		Implementación de propuesta heurística			
		Implementación de propuesta metaheurística			
		Diseño y orden de complejodad de los algoritmos propuestos			
5	**Publicaciones de doctorado**	Conferencia Keiser University			
		Papers para publicación nacional - primera ronda			
		Papers para publicación nacional - segunda ronda			
6	**Desarrollo de tesis doctoral**	Mejoras en el marco teórico			
		Mejoras en los resultados			
		Predefensa y defensa			

FASE		DETALLES	T8
			octubre 2023 / noviembre 2023 / diciembre 2023
1	Idea de tesis	Propuesta de idea a coordinacion	
		Propuesta de idea al tutor	
		Elaboración del bosquejo inicial	
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR	
		Elaboración de resumen para XXXIII SIMMAC 2022	
		Participacion en evento XXXIII SIMMAC 2022	
3	Desarrollo del protocolo tesis maestría	Estado del arte	
		Marco teórico	
		Resultados	
		Defensa de protocolo	
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos	
		Implementación de los algoritmos Metaheuristicos	
		Implementación de propuesta heurística	
		Implementación de propuesta metaheurística	
		Diseño y orden de complejodad de los algoritmos propuestos	
5	Publicaciones de doctorado	Conferencia Keiser University	
		Papers para publicación nacional - primera ronda	
		Papers para publicación nacional - segunda ronda	
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico	
		Mejoras en los resultados	
		Predefensa y defensa	

FASE		DETALLES
1	Idea de tesis	Propuesta de idea a coordinacion
		Propuesta de idea al tutor
		Elaboración del bosquejo inicial
2	Publicaciones de maestría	Inscripcion en evento de XXXIII SIMMAC 2022 UCR
		Elaboración de resumen para XXXIII SIMMAC 2022
		Participacion en evento XXXIII SIMMAC 2022
3	Desarrollo del protocolo tesis maestría	Estado del arte
		Marco teórico
		Resultados
		Defensa de protocolo
4	Diseño y Programación de los Algoritmos de la tesis doctoral	Implementación de los Algoritmos Heurísticos
		Implementación de los algoritmos Metaheuristicos
		Implementación de propuesta heurística
		Implementación de propuesta metaheurística
		Diseño y orden de complejodad de los algoritmos propuestos
5	Publicaciones de doctorado	Conferencia Keiser University
		Papers para publicación nacional - primera ronda
		Papers para publicación nacional - segunda ronda
6	Desarrollo de tesis doctoral	Mejoras en el marco teórico
		Mejoras en los resultados
		Predefensa y defensa