UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE ALGORITMOS SUBÓPTIMOS PARA REORDERING EN SÍNTESIS DE CIRCUITOS INTEGRADOS PARA SYNOPSYS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DIEGO ALBERTO RUIZ RAMIREZ

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

PROFESOR CO-GUÍA:
ALEJANDRO HEVIA ANGULO

COMISIÓN:
TOMÁS VERA

SANTIAGO DE CHILE
2024

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACION
POR: DIEGO ALBERTO RUIZ RAMIREZ
FECHA: 2024
PROF. GUÍA: GONZALO NAVARRO BADINO

IMPLEMENTACIÓN DE ALGORITMOS SUBÓPTIMOS PARA REORDERING EN SÍNTESIS DE CIRCUITOS INTEGRADOS PARA SYNOPSYS

La industria de los circuitos integrados experimenta un crecimiento desenfrenado y desempeña un papel fundamental en el avance tecnológico a nivel mundial. Synopsys, empresa líder en la industria, emplea la técnica de *Scan Testing* para verificar la funcionalidad de los circuitos integrados después de la fabricación, a través del proceso de *Scan Insertion*. Este proceso implica la incorporación de circuitos adicionales, incluida la conformación de una cadena entre componentes del circuito. En particular, el componente esencial de *Scan Insertion*, conocido como *Reordering*, se centra en optimizar la conectividad de esta cadena, buscando minimizar su largo. Este desafío se aborda como una instancia particular del Problema del Vendedor Viajero, reconocido por su complejidad NP-Completo.

Aunque los algoritmos actuales de Synopsys son funcionales, tienen un amplio margen de mejora y distan del estándar por el cual se guían las soluciones propuestas en la literatura académica. Tras revisar heurísticas desarrolladas en la academia para el Problema del Vendedor Viajero, se identifica el algoritmo Lin-Kernighan con mejoras de Keld Helsgaun como prometedor para implementar en Synopsys, a pesar de los desafíos en la implementación.

La implementación del algoritmo muestra resultados notables en la mejora de la optimalidad y tiempos de ejecución prometedores. Se considera que el nuevo algoritmo sea adoptado como el estándar en la herramienta de diseño de Synopsys, allanando además el camino a futuros desarrollos, al mismo tiempo que haciendo hincapié en el uso de estándares académicos y la conversión del problema de *Reordering* de Synopsys a una versión simétrica más estudiada, lo que resulta en una mayor amplitud de herramientas utilizables para resolver el problema.

Dedicado a mi madre,
la fuente primordial de todos mis logros.

Agradecimientos

Mi gratitud más profunda se dirige primero y ante todo hacia mi madre, la piedra angular indiscutible de mi crecimiento y éxito. Sus influencia y profundas enseñanzas han sido la semilla que ha guiado cada paso en mi viaje, orientándome siempre hacia los valores del trabajo arduo, el esfuerzo constante y la entrega total. A través de ella, he aprendido a apreciar la vida y a comprender que los frutos que disfruto hoy son el resultado directo de cultivar la virtud en cada paso que doy.

Le agradezco, además, por brindarme las condiciones ideales para centrarme en mi desarrollo personal e intelectual, aportando con todo su esfuerzo mental y físico. El tesoro más preciado que he construido a lo largo de mi vida son mis ideas, y esta riqueza se ha gestado gracias a la semilla intelectual, moral y ética que mi madre sembró en mí, regada constantemente por las condiciones ideales que ella misma propició.

Agradezco a la educación pública, un pilar esencial en mi desarrollo, que ha permitido mi crecimiento desde el Instituto Nacional hasta Beauchef en la Universidad de Chile. Mi gratitud se extiende a todos aquellos que, a lo largo de la historia de nuestro país, han trabajado incansablemente para hacer posible que acceder a las mejores instituciones educativas de Chile y Latinoamérica sea posible, sin que nunca me hayan exigido un pago por ello.

A mis profesoras de colegio, la Sra. María Jesús Barrientos de biología y la Sra. Jennifer Tambley de filosofía, quienes han dejado huellas imborrables en mi vida. La primera me mostró el vasto mundo intelectual que aguarda a quienes buscan crecer, mientras que la segunda me introdujo al placer del esfuerzo en la naturaleza, revelándome las lecciones valiosas que se esconden en desafiar los límites.

A mis amigos, auténticos pilares en mi vida, les agradezco de corazón. A Francisco "Chiscorey" por la amistad más genuina y pura; a Sebastián, por recordarme siempre la existencia de personas tan admirables y a la vez tan cálidas en el mundo. Y, por supuesto, a todos los queridos amigos que conformamos "La Lista": Asunción, Albani, Bruno, Cebolla, Cholo, Coti, Champi, Julia, Konrad, Pablito Skewes, Pau, Roberto, Sebita, Tomi y Vale. Con ustedes he compartido momentos fundamentales que me brindan una gratificación personal enorme. Hemos vivido tantos momentos hermosos juntos en tantos sentidos posibles; nuestro grupo es algo realmente especial, los quiero mucho.

Finalmente, agradezco a Synopsys por brindarme la oportunidad de unir la industria con la investigación a través de mi memoria. En especial, quiero expresar mi gratitud a Nicolás Rosso; gracias por siempre confiar en mí y en mi trabajo.

Tabla de Contenido

Índice de Tablas

Capítulo 1

Introducción

El circuito integrado es el instrumento fundamental en el desarrollo tecnológico. Su industria es pujante y se encuentra en constante e incansable competencia en busca de mejores optimizaciones, algoritmos, resultados, eficiencias y potencias.

En este contexto, Synopsys se posiciona como la empresa líder en soluciones de software para el diseño de circuitos integrados. La empresa ofrece una amplia gama de productos y servicios para ayudar a los ingenieros de diseño y de verificación de circuitos integrados a mejorar la calidad, el rendimiento y la seguridad de sus productos, así como a reducir los costos y el tiempo de desarrollo.

Una funcionalidad y problema particular en el diseño de circuitos integrados consiste en posibilitar la prueba de los mismos para verificar que fueron manufacturados correctamente. Este constituye un desafío de diseño, ya que es necesario desarrollar capacidades físicas en el circuito integrado que permitan esta funcionalidad.

En particular, en Synopsys, el equipo de "Design for Testability" (DFT), se encarga de desarrollar, optimizar y mejorar soluciones que aborden este problema.

Una de las técnicas utilizadas por DFT se conoce como *Scan Testing*. Esta técnica consiste en poner a prueba un circuito secuencial controlando su estado, luego ejecutando diversas combinaciones de entradas y comparándolas con los salidas esperados. De esta manera, se verifica el correcto funcionamiento de las componentes secuenciales y combinacionales del circuito integrado.

Para llevar a cabo lo mencionado, es necesario agregar una circuitería adicional, específicamente, un cableado alternativo que posibilite la manipulación del estado de los *flip-flops* que implemetan el estado. Esto implica la creación de una cadena de *flip-flops* , y para ello se requiere un nuevo cableado que conecte cada uno de los *flip-flops* que se desee forzar.

La decisión del orden en el cual conectar de forma óptima cada uno de los *flip-flops* no es trivial. Concretamente corresponde al proceso conocido como *Reordering* y, en abstracción, corresponde al conocido problema NP-Completo denominado "Problema del Vendedor Viajero".

Actualmente, Synopsys dispone de heurísticas relativamente simples y derivadas de *Nearest Neighbors* y 2-opt, heurísticas bien conocidas con las cuales aborda el problema del vendedor viajero.

Sin embargo, estas heurísticas han sido desarrolladas exclusivamente dentro de Synopsys y han sido comparadas únicamente con otras heurísticas internas de la misma empresa; esto se aleja de los conocimientos académicos, ya que dichas heurísticas han sido mejoradas principalmente a través de métodos de prueba y error.

Esta metodología de desarrollo ha conducido a la manifestación de comportamientos forzados en el software. Además, las pruebas realizadas en los algoritmos desarrollados por Synopsys hasta la fecha no son comparables en eficiencia y resultados con los de la literatura académica. Esto se debe a que el programa no permite la utilización de casos de prueba estándar, los cuales son empleados por cada algoritmo propuesto en la literatura.

La simplicidad de las heurísticas desarrolladas y su baja relación respecto a la investigación más avanzada en torno a heurísticas desarrolladas en la industria, ofrecen una enorme oportunidad de mejora. Es altamente probable que pueda mejorarse su proceso de *Reordering*, tanto en términos de tiempo y estandarización respecto a la academia. Además, existe la posibilidad de mejorar la calidad de la solución a la vez que reducir la excesiva complejidad de los pasos. Esto abre la puerta a la implementación de heurísticas, o combinaciones de las mismas, más estudiadas y con resultados comparables al conjunto de todas las demás heurísticas desarrolladas, no solo las de Synopsys.

La obtención de tiempos más eficientes genera un claro efecto positivo para el cliente al reducir su tiempo de espera en el ciclo de desarrollo de circuitos integrados. Por otro lado, un mayor grado de optimalidad conlleva una multitud de beneficios adicionales, tales como la reducción en el consumo energético y la disipación de energía, así como la optimización del área utilizada por el circuito, entre otros (TODO: explicar por que y dar citas).

Como respuesta a esta problemática, se propone lo siguiente: una revisión extensiva de la literatura con el objetivo de caracterizar el problema de *Reordering* y al mismo detectar oportunidades para una mejor resolución de dicho problema. Esta revisión exhaustiva tiene como objetivo proporcionar el conocimiento necesario para abordar el problema de la manera más precisa posible, reemplazando las soluciones ininteligibles actualmente implementadas y aprovechando al máximo cada uno de los puntos identificados a través de la investigación.

Como resultado de esta revisión, se tiene como objetivo determinar un algoritmo a implementar, utilizando el conocimiento extraídos de los múltiples algoritmos investigados. La meta es implementar un algoritmo que recoja cada uno de los beneficios presentes en los diferentes documentos estudiados, con la aspiración teórica de que sea la mejor opción para mejorar el proceso de *Reordering* de Synopsys.

Luego, se planifica implementar el algoritmo, cuidando la simplicidad y extensibilidad del mismo. Esto permitirá su mejora y cambio a futuro de manera eficiente y simple.

Finalmente, se evaluarán los resultados del algoritmo desarrollado, midiendo su desempeño

según el estándar académico y comparándolo tanto con los algoritmos actuales de Synopsys como con otros algoritmos desarrollados en la industria.

Capítulo 2

Estado del arte

2.1. Design for Testability

El diseño de circuitos integrados abarca una serie de pasos complejos, representando uno de los procesos más intrincados en la historia de la ingeniería. En esta complejidad reside la crucial tarea de verificar el correcto funcionamiento del circuito integrado una vez impreso, especialmente teniendo en cuenta que estos dispositivos se desarrollan a una escala nanométrica y contienen millones de componentes [1]

En la evolución de la tecnología de circuitos, se evidencia un continuo aumento en el número de compuertas, un fenómeno que se magnifica con cada nueva generación. Este crecimiento, aunque trae consigo mejoras en el rendimiento de los chips, también introduce una complejidad considerable en la determinación, de manera costo-efectiva, si un chip específico ha sido manufacturado correctamente.

A lo largo de la historia del desarrollo de los circuitos integrados, se han explorado diversas alternativas de prueba, dada la creciente necesidad de evaluar su funcionamiento. En 1983, Williams, en su artículo [1], ya señalaba la urgencia de una disciplina para abordar este problema, proponiendo la simulación de pruebas como una técnica estándar, práctica que aún persiste en empresas líderes como Synopsys. "La prueba formal ha sido, hasta la fecha, prácticamente imposible, y la simulación de fallas, respaldada por numerosas pruebas, se erige como una de las alternativas más sólidas. En este contexto, las técnicas agrupadas bajo el término "Design for Testability" se presentan como una aproximación general para abordar este desafío".

Vale mencionar también que el proceso de prueba del correcto funcionamiento de la manufactura se desarrolla bajo altísimas presiones de tiempo, impulsado no solo por la inherente competitividad de la industria, sino también por la necesidad, según se destaca en [2], de realizar este proceso en las etapas finales del diseño, lo que tiende a ejecutarse bajo extrema presión temporal [2].

Scan Testing: En la disciplina de DFT, se han desarrollado diversas técnicas para abordar el desafío de probar la correcta manufactura del circuito integrado. Entre ellas, destaca la técnica conocida como *Scan Testing*, la cual constituye el foco central de esta memoria.

Como principio fundamental para comprender esta técnica, resulta crucial reconocer que los circuitos a testear son secuenciales, es decir, cuentan tanto con una componente secuencial como combinacional. Mientras que la componente combinacional tiene un comportamiento funcional, produciendo una salida consistente ante una misma entrada, la componente secuencial se comporta abstractamente como una máquina de estados. En términos concretos, esta máquina de estados corresponde al efecto agregado de un conjunto de *flip-flops*. Cada combinación de valores en los *flip-flops* da origen a un estado.

La forma de probar el correcto funcionamiento de este circuito secuencial que presenta *Scan Testing* implica probar exhaustivamente la componente combinacional con una enorme cantidad de combinaciones de entradas de las cuales se conoce su salida correcta.

El problema radica en que la salida del circuito en su totalidad no es funcional debido a la presencia de la componente secuencial. Por ende, para realizar pruebas, es necesario conocer el estado particular en que se están llevando a cabo y, aún más, es necesario poder forzar el estado para verificar si se comporta correctamente en cada uno de los estados posibles.

Recapitulando, es necesario probar el circuito con una multitud de combinaciones posibles de entradas, al mismo tiempo que se fuerza un estado conocido.

Para lograr cada uno de estos objetivos, se llevan a cabo los siguientes procedimientos:

Generación de combinaciones de prueba: Esto se logra utilizando una máquina ATE ("Automatic Test Equipment") para generar una enorme cantidad de casos de prueba y probar el chip sobre ellos [3].

Forzado de estado del circuito: Para esto se emplea la técnica denominada *Scan Insertion*, la cual consiste en agregar circuitería adicional a cada *flip-flop*, introduciendo un multiplexor y una pista conductora adicional. Este multiplexor tiene dos opciones, denominadas modo funcional y modo de pruebas. Mientras que el modo funcional representa el funcionamiento normal del circuito, el modo de pruebas tiene como objetivo forzar el estado de los *flip-flops* [4, 5].

La entrada del modo de pruebas se implementa mediante un *shift register*. Un *shift register* es un circuito que, tras cada ciclo de reloj, genera un desplazamiento serial de datos a través de posiciones de almacenamiento, en este caso, los *flip-flops*. El desplazamiento serial contiene el estado deseado de para cada uno de los *flip-flop*.

Para transmitir la señal serial, se construye una cadena de cableado, concretamente, pistas conductoras en el silicio, partiendo desde el *shift register* y pasando por cada *flip flop* hasta llegar al último de ellos. Luego, tras múltiples ciclos de reloj, la información del estado de cada *flip flop* llega desde el shift register a cada uno de ellos, forzando así el estado.

Para que cada *flip-flop* reciba esta señal, es necesario que el multiplexor de cada *flip flop* esté configurado en modo de prueba. A cada uno de estos pares multiplexor-*flip-flop* se le denomina *scan-flop*.

Una vez el estado de cada *flip-flop* se encuentre determinado, se procede a probar el circuito en ese estado particular, ingresando enormes cantidades de combinaciones de entrada, comparando entonces, su salida generada con una salida esperada conocida para cada caso de prueba. En base al grado de diferencia entre ambos resultados se determina si el circuito sigue el comportamiento correcto y esperado o si tiene fallas de manufactura.

La introducción de esta nueva lógica de prueba simplifica el control y la observación de la variable de estado del circuito, permitiendo la verificación de la porción combinacional del circuito [5]. Esto se logra sin interferir con el funcionamiento normal, simplemente activando cada multiplexor para que recoja la señal funcional.

Tras probado exhaustivamente el comportamiento de la componente combinacional dado un estado, se prosigue mediante el *shift register* y la transmisión serial de datos entre *flip-flops*, a determinar un nuevo estado, sobre el cual nuevamente se prueban combinaciones de entrada de forma exhaustiva, buscando discrepancias entre solución resultante y solución esperada.

En resumen, *Scan Insertion* agrega nueva circuitería al circuito, en particular a los *flip-flops*, introduciendo una nueva señal a través de la cual, mediante un *shift register*, se fuerza el estado del circuito. Una vez forzado el estado del circuito, se prueban tantas combinaciones de entrada como sea posible, comparando sus salidas generadas con las salidas esperadas y determinando así si el circuito presenta fallas [1, 4].

A través de esta introducción a *Scan Testing*, se han presentado múltiples conceptos, cada uno con una denominación particular que permite referirse a ellos de manera precisa y específica. Estos conceptos son:

Scan Flop: Se denomina *scan flop* a cada uno de los pares multiplexor-*flip-flops* que pueden operar tanto en modo funcional, pudiendo recibir la señal funcional como la señal de prueba para forzar el estado del mismo [4].

Scan Chain: Se denomina *scan chain* al conjunto de conexiones entre *scan flops*, formando el cableado a través del cual la señal serial del *shift register* se desplaza a lo largo de los *scan flops*. Es relevante destacar que la conexión entre *scan flops* implica que el input de un *scan flop* recibe un cableado del output del *scan flop* anterior en la cadena [4].

En particular, es útil notar que, considerando el conjunto de *scan flops* a los cuales se busca forzar su estado, estos conforman un grafo. En este contexto, la *scan chain* corresponde en lo abstracto, a un camino hamiltoniano en dicho grafo, es decir, un camino que recorre cada uno de los vértices, pasando por cada vértice una y solo una vez.

Partición: En circuitos extensos con cientos de miles de *flip-flops*, a menudo se busca probar el funcionamiento de secciones que pueden ser aisladas funcionalmente del resto del circuito.

Para ello, en lugar de conectar todos los *scan flops* de un circuito completo, se realiza una partición del conjunto de *scan flops* en múltiples cadenas. Por ejemplo, podría haber un circuito encargado de la funcionalidad A, completamente independiente de otro circuito

encargado de la funcionalidad B. El interés radica en probar independientemente cada una de estas funcionalidades, formando una cadena con los *scan flops* del circuito A y otra cadena con los *scan flops* del circuito B.

Aunque ambas funcionalidades sean independientes, podrían ser parte de una misma sección del circuito total, permitiendo agruparlas. Una sería que un circuito A es "lava los platos en la casa", mientras que otro circuito B es "limpia los baños"; ambos son parte de la partición "Casa". Mientras que un circuito C es "escribe un informe" y otro circuito D es "rinde una evaluación", y ambos son parte de la partición "Universidad".

De manera similar, un circuito extenso se compone de múltiples subcircuitos, cada uno con su función específica. Por lo tanto, un gran circuito se puede describir como un conjunto de particiones, y cada partición a su vez como un conjunto de cadenas.

Esta noción se describe en [1] de la siguiente manera: "La partición, en el contexto de la prueba de circuitos, se define como la capacidad de desconectar una porción de una red de otra con el fin de facilitar el proceso de pruebas".

Proceso de *Reordering*: Ahora, comprendido el proceso de *Scan Testing*, esta memoria se centra particularmente en el proceso de *Scan Insertion*. Al abordar un circuito completo y extenso, surgen cuestiones clave. ¿Cómo particionar el conjunto masivo de todos los *scan flops* en particiones? Este proceso, conocido como *Partitioning*[1], es una área profunda, pero no es de interés para esta memoria. Supongamos que ya se ha dividido todos los *scan flops* en particiones. Ahora, ¿cómo se particiona el total de *scan flops* de cada partición en cadenas? Nuevamente, esto corresponde al proceso de *Partitioning*, e incluso más específicamente, al proceso de *Repartitioning*, que es iterativo y busca formas óptimas de conformar cadenas dentro de una partición, pero esto tampoco es relevante para esta memoria.

Supóngase que ya fueron ejecutados los procesos de *Partitioning* y *Repartitioning* y que ya todos los *scan flops* de un gran circuito fueron distribuidos en particiones y, además, dentro de cada partición, se han distribuido sus *scan flops* en cadenas.

Es evidente que ahora, dentro de cada cadena, debe determinarse una *scan chain* que conecte cada uno de los *scan flops* en un camino hamiltoniano desde un *shift register* [1].

Y aquí surge la pregunta fundamental que aborda esta memoria.

Es claro que debe formarse una cadena de *scan flops*, pero ¿en qué orden? Por ejemplo, si se tiene n *scan flops* dentro de alguna cadena y enumeramos cada uno de los *scan flops* desde 1 hasta n, podemos generar una secuencia que define el orden. Con esto, se tienen $n!$ opciones para conformar esta cadena. Entonces, surge la pregunta: entre esas $n!$ opciones, ¿cuál es la mejor? Para esto, primero debe establecerse qué es lo deseable y qué es lo indeseable. A lo largo de la historia del proceso de *Scan Testing*, se ha afirmado que lo fundamental es la reducción en la longitud total del cableado [3]. En otras palabras, mientras más largo sea el cable total de una *scan chain*, peor. Buscandose una longitud de cableado pequeña.

Surge entonces el problema denominado *Ordering* y *Reordering* de *scan chains* proceso

que tiene por fin, encontrar el ordenamiento óptimo, tal que el largo de la *scan chain* sea minimizado [3].

Para abordar este problema, se examinan algunas abstracciones en la siguiente sección.

2.2. Problema del Vendedor Viajero

Considérese a un vendedor de presenta un producto, con la aspiración de comercializarlo en varias ciudades del mundo. En el transcurso de su gira, debe visitar cada una de estas ciudades, procurando visitar cada una solo una vez y, al completar este itinerario, desea retornar a su punto de origen. Para abordar este desafío de la forma más óptima posible, es crucial tener en cuenta los costos asociados con los desplazamientos entre las ciudades. En este contexto surge la pregunta para el vendedor. ¿Cuál es la ruta más económica que puede seguir para cumplir su objetivo, dada la posibilidad de trasladarse desde cualquier ciudad a otra? Este problema corresponde al clásico Problema del Vendedor Viajero, de ahora en adelante también TSP, por sus siglas en inglés, uno de los problemas de optimización combinatorial más reconocido y estudiado y clasificado como problema NP-Completo[6].

La clase de problemas NP-Completo es una clase de problemas de alta dificultad, cuya complejidad computacional de resolución es desconocida pero probablemente exponencial. Los miembros de esta clase se encuentran relacionados, en cuanto encontrar una solución polinomial para uno de ellos, implicaría que también existe una para todos los demás problemas de la clase. Sin embargo es de general consenso que esto no es posible. En conclusión, toda intención de construir un algoritmo polinomial para resolver el Problema del Vendedor Viajero, muy probablemente fallará [6].

El Problema del Vendedor Viajero ilustra claramente su escalada en complejidad a medida que aumenta el número de ciudades consideradas en la ruta. Visualicemos esta escalada por ejemplo con el caso de 4 ciudades, para el cual existen $\frac{(4-1)!}{2} = 3$ posibles *tour*, esto es ínfimo. Sin embargo, al incrementar el número de ciudades a 10, la cantidad de *tours* posibles se dispara a 3.628.800, mostrando un crecimiento exponencial en las opciones disponibles [6].

En general, para n ciudades, el número de posibles *tours* en el Problema del Vendedor Viajero se expresa como $\frac{(n-1)!}{2}$[6].

La magnitud de esta complejidad se vuelve aún más impactante al considerar el caso de 60 ciudades, donde el número de posibilidades alcanza 60!, es decir, aproximadamente 8.3×10^{81} *tours* posibles. Esta cifra es notablemente superior al número estimado de átomos en el universo, que se sitúa en 8×10^{80}. Este contraste resalta la vasta gama de posibilidades que debe explorar el Problema del Vendedor Viajero a medida que el número de ciudades aumenta, evidenciando la complejidad inherente y la dificultad computacional asociada con este desafío de optimización combinatorial [6].

Debido a su complejidad factorial, resulta impracticable realizar una búsqueda exhaustiva de cada *tour* posible, calcular sus costos y elegir el mejor. Esta dificultad se intensifica rápidamente a medida que n aumenta, como lo evidencian los sorprendentes números asociados

con 60 ciudades.

Respecto a definiciones más precisas, este problema tiene múltiples definiciones equivalentes, destacándose para esta memoria las tres siguientes.

En términos de conceptos de teoría de grafos, cada ciudad corresponde a un vértice de un grafo, y se puede viajar desde cualquier ciudad a otra. Un viaje de una ciudad a otra se caracteriza como una arista que une ambos vértices, y además, el costo de viaje se asocia a cada arista del grafo. Se tiene que el Problema del Vendedor Viajero corresponde a encontrar un camino hamiltoniano de costo mínimo en un grafo completo [6].

Matemáticamente, en términos de representar el grafo según su matriz de costos, se tiene la siguiente definición: Dada una matriz de costos $C = (c_{i,j})$, donde $c_{i,j}$ representa el costo de ir desde la ciudad i a la ciudad j, con $i, j = 1, ..., n$. Encontrar una permutación $(i_1, i_2, i_3, ..., i_n)$ de enteros desde 1 hasta n tal que minimiza la cantidad $c_{i_1 i_2} + c_{i_2 i_3} + ... + c_{i_n i_1}$ [7].

Adicionalmente, representando a un *tour* como un conjunto de aristas, se busca el subconjunto de aristas cuya suma de costos sea mínima, y este conjunto cumpla la propiedad de conformar un ciclo hamiltoniano en el grafo [6].

Mientras que la definición en términos de conceptos de teoría de grafos permite mantener la intuición, la descripción según la matriz de costos será útil próximamente para explicar soluciones para este conocido problema combinatorial.

En términos de soluciones computacionales desarrolladas, la complejidad inherente del problema TSP se ilustra aún más por los desafíos enfrentados por Concorde, uno de los *solver* más avanzados de TSP. En un caso notable, se requirieron 155 días para calcular la solución óptima para un conjunto de 11,849 ciudades, como se indica en su página web. Este ejemplo concreto pone de manifiesto la imposibilidad práctica de resolver el Problema del Vendedor Viajero de manera exacta para instancias de gran escala.

Dada entonces la naturaleza NP-hard del problema TSP, nos enfrentamos a la necesidad de abordarlo mediante heurísticas y algoritmos aproximados. Estos enfoques ofrecen soluciones eficientes en tiempo para conjuntos de datos considerablemente grandes, aunque no garantizan la obtención de la solución óptima. La resolución práctica del Problema del Vendedor Viajero, por lo tanto, involucra, en sus versiones más *naive*, un fuerte *trade-off* entre la complejidad computacional y la precisión de la solución, haciendo que las heurísticas sean herramientas valiosas en la búsqueda de rutas eficientes en escenarios del mundo real frente a soluciones complejas de complejidades infactibles [8].

Pensando en el Problema del Vendedor Viajero desde su enunciado como un problema de teoría de grafos, es necesario definir las características principales de este grafo.

Caso simétrico: En el Problema del Vendedor Viajero simétrico (TSP), se establece una condición específica en la que el costo de desplazarse desde la ciudad A hasta la ciudad B es idéntico al costo de realizar el recorrido inverso, es decir, de la ciudad B a la ciudad A [7].

Matemáticamente, esto corresponde a la propiedad $\forall i, j, c_{ij} = c_{ji}$.

Históricamente, el Problema del Vendedor Viajero simétrico ha sido objeto de una amplia variedad de investigaciones y desarrollos de heurísticas. Se han explorado diferentes *trade-offs* entre la complejidad de implementación, el tiempo de ejecución y la búsqueda de soluciones óptimas. La simplicidad derivada de la simetría permite abordar el problema de manera más eficiente y con mayor precisión en términos de optimización [8].

Caso asimétrico: En el Problema del Vendedor Viajero asimétrico (ATSP), se presenta una condición en la cual el costo de desplazarse desde una ciudad A hasta una ciudad B no necesariamente es igual al costo de realizar el recorrido inverso, de la ciudad B a la ciudad A. Esta asimetría en los costos de desplazamiento entre las ciudades añade una capa adicional de complejidad al problema.

Matemáticamente, corresponde a la propiedad $\forall i, j, c_{ij} \neq c_{ji}$

El caso asimétrico del Problema del Vendedor Viajero ha recibido menos atención en comparación con su contraparte simétrica en términos de estudios y análisis

Respecto a la descripción del costo de las aristas del grafo, estos costos pueden representar distintas nociones, como tiempo, dinero o distancia, entre otras.

En cuanto a su uso, este problema corresponde a una abstracción que se encuentra presente en una variedad de aplicaciones. Tal como indica Helsgaun, "La importancia de TSP viene de una amplia riqueza de aplicaciones, muchas de las cuales, aparentemente no tienen nada que ver con el viaje a través de rutas" [7].

Algunas de sus aplicaciones incluyen, por ejemplo, el ruteo de vehículos, cristalografía, control de robots, planificación de una bodega y diseño de circuitos [9].

2.3. Relación entre el Problema del Vendedor Viajero y el Proceso de *Reordering*

Freuer y Kroo, en su trabajo de 1983 [10], fueron los primeros en identificar la fuerte relación y notable analogía entre el proceso de *Reordering* de una *scan chain* y el clásico TSP. Esta observación crucial permitió abstraer el problema de *Reordering* como una aplicación específica del TSP, brindando una nueva perspectiva para enfrentar este desafío y además, poder utilizar la gran cantidad de investigación que se ha realizado en torno al TSP a lo largo de la historia.

Sin embargo, hay algo que tener en cuenta. Mientras que el TSP resuelve el problema de encontrar el ciclo hamiltoniano de menor costo, el problema de *Reordering* no construye un ciclo, sino un camino. Si tenemos en cuenta que una *scan chain* conoce su inicio en el *shift register* y conoce su final en el último *scan flop*, el cual es definido como tal, entonces

simplemente basta con realizar una conexión virtual desde el último *scan flop* al *shift register*, cuyo largo sea 0 y la definición de camino y ciclo resultan equivalentes [1, 4, 5].

Describiendo entonces la analogía desde su expresión más intuitiva y menos formal como un viajero, se puede decir que un *shift register* quiere viajar por cada *scan flop* una vez, forzando (vendiendo) su estado, y luego volver a su posición inicial.

En particular, concretizando esa analogía, cada uno de los *scan flops* corresponde a un vértice, mientras que la pista conductora que une dos *scan flops* corresponde a la arista que une sus vértices respectivos. Dado que los *scan flops* están ubicados en el espacio bidimensional, y la cantidad que se quiere minimizar es el largo de la pista conductora, entonces el costo ha de ser el largo de cada pista que une los *scan flops*. Y ya que las pistas se disponen en patrones cuadrados, el largo de una pista conductora entre dos *scan flops* al ir desde el *scan flop* A hasta el *scan flop* B corresponde a la función matemática de distancia Manhattan entre el out del *scan flop* A y el in del *scan flop* B. Ya que cada *scan flop* tiene sus conexiones in y out en distintas ubicaciones en el plano bidimensional, entonces el problema es naturalmente asimétrico, en cuanto ir desde el out de A al in de B es distinto en instancia Manhattan a ir desde el out de B al in de A.

Esta conceptualización ofrece una perspectiva interesante sobre cómo problemas aparentemente distintos, como el TSP y el *Reordering* de cadenas de exploración en *Scan Testing*, comparten similitudes fundamentales. Al abordar el problema de *Reordering* como una instancia asimétrica del TSP, se facilita la aplicación de heurísticas y estrategias derivadas del vasto conjunto de conocimientos desarrollados para el TSP, aportando una mayor comprensión y eficiencia al proceso de *scan chain*.

Luego, al lograr definir una buena heurística para el Problema del Vendedor Viajero asimétrico, la cual cumple los intereses de Synopsys, es decir, menor costo en un tiempo razonable, estaremos al mismo tiempo definiendo una buena heurística para el problema de *Reordering*.

Para evaluar qué tan buena es una heurística en términos de costo, necesitaremos definir un algoritmo sobre casos de prueba de TSP, el cual nos entregue un valor de optimalidad, ya sea el óptimo real como una cota inferior cercana al mismo. A su vez, tras haber definido una métrica absoluta de la calidad de un algoritmo, se exploran múltiples heurísticas desarrolladas a través de la historia de este problema.

2.4. Algoritmos para la resolución y evaluación de TSP

En el amplio espectro de estrategias destinadas a abordar el desafiante Problema del Vendedor Viajero, los algoritmos desarrollados se pueden clasificar dentro de cuatro clases principales, cada una con un enfoque distintivo para afrontar la resolución del problema o evaluar sus resultados. Respecto a aquellas diseñadas para resolver el problema TSP, existe un conjunto de opciones algorítmicas que abarca desde métodos exactos, que buscan la solución óptima mediante un examen exhaustivo, hasta enfoques heurísticos que priorizan la eficiencia computacional para instancias más grandes.

En este capítulo, se delinea brevemente cada una de las cuatro clases de algoritmos para TSP: los algoritmos exactos, que se sumergen en la búsqueda rigurosa de la solución óptima; los algoritmos para obtener cotas inferiores, que establecen referenciales para evaluar la calidad de las soluciones; los algoritmos constructivos, que construyen soluciones paso a paso; y, finalmente, los algoritmos de mejora iterativa, que buscan refinar y ajustar las soluciones existentes para lograr mejoras sustanciales. A través de esta clasificación, se explorarán las distintas estrategias empleadas para enfrentar la complejidad del TSP y obtener soluciones eficientes en tiempo y cercanas a la óptima en situaciones del mundo real.

2.4.1. Algoritmos exactos

Estos algoritmos se distinguen por su enfoque exhaustivo, aspirando a encontrar la solución óptima al explorar de manera sistemática todas las posibles combinaciones de rutas. Aunque este método garantiza la certeza de obtener la mejor solución posible, su implementación se vuelve inútil para enfrentar la gran mayoría de las aplicaciones de TSP debido a la complejidad computacional exponencial de la misma, generando algoritmos que requieren un tiempo de ejecución abismal.

El altísimo costo de estos algoritmos vuelve infactible su uso en aplicaciones, las cuales por norma general requieren rapidez de respuesta, sin embargo considerarlos permite definir un conjunto de casos de prueba en los cuales se conozca el óptimo, dando entonces un estándar con que comparar algunas instancias de TSP a las cuales se les haya dedicado el tiempo suficiente y capacidad de procesamiento para determinar sus óptimos.

A continuación se listan algunos de los más fundamentales:

Algoritmo de Held-Karp: El algoritmo de Held-Karp, propuesto por Held y Karp en su artículo de 1970, [11], utiliza la programación dinámica para obtener de manera exacta la solución al TSP.

La clave de este algoritmo es su formulación recursiva y la utilización de subproblemas más pequeños para construir la solución global. La idea central es calcular la longitud óptima de todos los subconjuntos de ciudades que incluyen la ciudad de inicio. A través de la memorización, evita recalcular soluciones para subproblemas ya resueltos, mejorando así la eficiencia computacional.

El algoritmo de Held-Karp tiene una complejidad temporal de aproximadamente $O(n^2 * 2^n)$, donde n es el número de ciudades.

Algoritmo *Branch and Bound*: Esta técnica se basa en la idea de explorar selectivamente el espacio de soluciones para identificar y descartar ramas que no conducen a soluciones óptimas, evitando así un examen exhaustivo de todas las posibilidades [8].

El procedimiento se inicia mediante la formulación de una solución inicial, que se mejora iterativamente mediante la exploración de ramas adicionales. En cada paso, se selecciona una ciudad no visitada y se generan dos subproblemas: uno que considera la inclusión de la ciudad en el *tour* y otro que la excluye. Estos subproblemas se resuelven recursivamente, y

la exploración continúa hasta que se alcanza una solución completa.

Una parte fundamental del algoritmo de *Branch and Bound* es la poda de ramas, donde se descartan subproblemas que no pueden conducir a una solución óptima. Esto se logra comparando las soluciones parciales con la mejor solución conocida hasta el momento. Si se identifica que un subproblema no puede superar la mejor solución actual, se omite su exploración, reduciendo así la complejidad del algoritmo.

Aunque el algoritmo de *Branch and Bound* encuentra la solución exacta al TSP, su desempeño puede depender significativamente de la eficacia de las estrategias de poda implementadas y del tamaño de la instancia del problema, viéndose en promedio un alto costo computacional.

Sin embargo, el algoritmo de *Branch and Bound* es frecuentemente usado para encontrar soluciones óptimas para un conjunto de casos de prueba, pudiendo ser fácilmente aplicado para resolver TSP, tanto en su versión simétrica como asimétrica [8].

A pesar de estas limitaciones, los algoritmos exactos continúan siendo esenciales para establecer los estándares de calidad en la resolución del TSP y para comprender la naturaleza intrínseca de este desafío combinatorio.

2.4.2. Algoritmos de cota inferior

En la búsqueda de soluciones que den resultados de buena calidad para el TSP, los algoritmos de cota inferior desempeñan un papel clave, aún más cuando se quiere determinar un estándar respecto a un caso que no ha sido considerado para ser resuelto por los algoritmos exactos o que es de uso privado [8].

Estos algoritmos, de tiempo de ejecución aceptable, de costo polinomial en el caso promedio, permiten obtener una cota inferior de una instancia dada de TSP al ser aplicados sobre ellos. O sea, permiten determinar que ninguna solución obtendrá una solución menor a ese valor, permitiendo entonces conocer con cierto nivel de mejor certeza, qué tanto se puede mejorar, aún más, un algoritmo que busque solucionar TSP, permitiendo la evaluación en términos absolutos de un conjunto de algoritmos sobre cualquier problema que se quiera.

Cota inferior de Held-Karp: Introducido por Held y Karp en su influyente trabajo de 1970 [11], el algoritmo de cota inferior de Held-Karp se destaca como una herramienta esencial en la resolución de TSP. Diseñado para proporcionar una estimación mínima de la longitud del *tour*, esta cota inferior, denominada de Held-Karp, exhibe una eficacia notable al situarse usualmente un 0.8 % por debajo del valor óptimo del *tour*. No solo presenta un grado de optimalidad cercano al real, sino que también garantiza que la menor cota inferior alcanzable mediante Held-Karp sea al menos un 2/3 de la longitud del *tour* óptimo, como se demostró en estudios posteriores en [12].

La base teórica de la cota inferior de Held-Karp reside en la solución del problema de programación lineal relajada derivado de la formulación de programación entera del TSP [12]. Esta solución, obtenida a través de métodos de programación lineal, constituye una estima-

ción rigurosa y eficiente de la longitud mínima de la ruta. Cabe destacar que el procedimiento para hallar esta solución se realiza en tiempo polinómico, gracias a la aplicación del Método de Simplex y un algoritmo de restricción por separación polinomial [12].

En la práctica, la cota inferior de Held-Karp se convierte en una referencia fundamental para evaluar el rendimiento de diversas heurísticas diseñadas para resolver el TSP. La comparación de los resultados obtenidos por algoritmos heurísticos con respecto a esta cota inferior proporciona una medida esencial de la calidad y eficacia de las soluciones propuestas. En este sentido, la cota inferior de Held-Karp se erige como un estándar de referencia valioso en la búsqueda de soluciones eficientes y cercanas a la óptima para el Problema del Vendedor Viajero.

2.4.3. Algoritmos constructivos

En la búsqueda de encontrar soluciones de la máxima calidad posible en el menor tiempo, los algoritmos constructivos se centran en construir soluciones paso a paso, comenzando con un conjunto inicial y expandiéndolo iterativamente para formar un *tour* [8].

Este enfoque constructivo ha sido ampliamente adoptado en múltiples aplicaciones donde se usa TSP, ya que es relativamente rápido y entrega soluciones relativamente buenas.

A continuación, se presentan algunos de los algoritmos más reconocidos de este grupo, mostrando sus resultados respecto a las mediciones estándares de cota inferior.

***Nearest Neighbor*:** Este algoritmo corresponde probablemente al constructivo más conocido y comúnmente utilizado para resolver TSP [8].

El enfoque consiste en, sucesivamente, a partir de un nodo, seleccionar el siguiente nodo a visitar basándose en la distancia más corta desde el nodo actual. Los pasos se pueden describir de la siguiente manera:

1. Seleccionar una ciudad aleatoria.
2. Encontrar la ciudad aún no visitada más cercana y agregarla.
3. ¿Hay alguna ciudad sin visitar aún? Si aún hay alguna, repetir desde el paso 2.
4. Volver a la primera ciudad.

Calificado como posiblemente la heurística más simple y directa, este algoritmo generalmente obtiene soluciones un 25 % por encima de la cota inferior de Held-Karp [8].

Algoritmo *Greedy*: La heurística Greedy construye gradualmente un *tour* seleccionando repetidamente la arista más corta y añadiéndola al *tour* siempre que no genere un ciclo con menos de n aristas o aumente el grado de cualquier nodo a más de 2. Por supuesto, no debemos agregar la misma arista dos veces. El algoritmo Greedy tiene una complejidad de $O(n^2 \log_2(n))$ [8].

Se describe esta heurística según los siguientes pasos:

1. Ordenar todas las aristas.
2. Seleccionar la arista más corta y añadirla a nuestro *tour* si no viola ninguna de las restricciones mencionadas anteriormente.
3. ¿Tenemos n aristas en nuestro *tour*? Si no, repetir el paso 2.

El algoritmo Greedy generalmente se mantiene dentro del 15-20 % de la cota inferior de Held-Karp [8].

Algoritmo de Christofides: La mayoría de las mejores heurísticas constructivas desarrolladas solo pueden asegurar ser una 2-aproximación, es decir, en el peor de los casos, obtener un *tour* que tenga el doble de largo que el *tour* óptimo. Sin embargo, el profesor Nicos Christofides llevó a cabo una extensión de uno de estos algoritmos, obteniendo teóricamente que el resultado es una 1.5 aproximación, es decir, en el peor de los casos, el *tour* obtenido tiene un largo 1.5 veces el del *tour* óptimo. Este método es ampliamente reconocido como la heurística de Christofides [8].

Los pasos de la heurística de Christofides corresponden a los siguientes [8]:

1. Construir un árbol de expansión mínima a partir del conjunto de todas las ciudades.
2. Crear un emparejamiento de peso mínimo en el conjunto de nodos con grado impar. Agregar el árbol de expansión mínima junto con el emparejamiento de peso mínimo resultante.
3. Crear un ciclo euleriano a partir del grafo combinado y recorrerlo tomando atajos para evitar nodos ya visitados.

El principal componente costoso de este algoritmo radica en la obtención del árbol de expansión mínima. En cuanto a los valores promedio de optimalidad que logra este algoritmo, generalmente se sitúan alrededor de un 10 % por encima de la cota inferior de Held-Karp [8].

Vale la pena señalar que, tal como se evidencia para el algoritmo de Christofides, a medida que se logran mejoras en los resultados, esto puede ir acompañado de una mayor complejidad en el diseño del experimento y, a su vez, una mayor complejidad en su implementación. Esto será una tendencia en adelante, en cuanto se busca obtener los mejores resultados posibles.

Respecto a otras heurísticas constructivas, existen múltiples alternativas para resolver el Problema del Vendedor Viajero. Sin embargo, las presentadas anteriormente son consideradas como las más prometedoras y ampliamente reconocidas.

2.4.4. Algoritmos de optimización local

Los algoritmos de optimización local se presentan como una alternativa diferente para obtener soluciones a TSP. Mientras que los algoritmos constructivos conforman un *tour* a partir de un conjunto inicial de vértices sin conexión, los algoritmos iterativos parten de un *tour* ya construido y lo mejoran cambiando aristas del *tour* por otras que mantengan la estructura del *tour* pero a un menor costo total [8].

Para este tipo de algoritmos, el enunciado del problema de TSP como encontrar el subconjunto de aristas que minimiza la función de costos y mantiene el invariante de *tour* resulta más claro, ya que estos algoritmos trabajan directamente con este conjunto, agregando y quitando elementos del mismo, es decir, aristas.

Este tipo de algoritmos se puede describir de manera general mediante la siguiente secuencia de pasos, como se señala en [13]:

1. Generar una solución factible pseudoaleatoria, es decir, definir un subconjunto de aristas que conformen un *tour*.
2. Intentar encontrar una versión mejorada al aplicar una transformación a la solución factible definida.
3. Si se obtiene una versión mejorada, reemplazar la versión factible por la mejorada y volver al paso 2.
4. Si no se puede encontrar una versión mejorada, entonces la solución actual es un óptimo local. Repetir desde el paso 1 hasta que el tiempo de ejecución definido se agote o se obtenga una solución suficientemente buena.

Mientras mejor sea una heurística de optimización local, su conjunto de óptimos locales tiende a disminuir, lo que, a su vez, amplía la variedad de *tour*s de inicio aleatorios que pueden converger hacia el óptimo global [13].

La diferencia principal entre cada uno de los distintos algoritmos que conforman esta clase radica principalmente en cómo definen el paso 2. La modificación más común implica el intercambio de aristas pertenecientes al *tour* actual por otras aristas, manteniendo la estructura del *tour* [13].

Veamos entonces a continuación algunos de los algoritmos de optimización local más reconocidos y ampliamente utilizados, tanto por sus resultados destacables como por sus tiempos de ejecución.

Algoritmo 2-opt: El algoritmo 2-opt destaca como el algoritmo de optimización local por excelencia, siendo el más ampliamente reconocido dentro de esta clase. Su popularidad se atribuye a su simplicidad y la capacidad de proporcionar buenos resultados, estableciendo un equilibrio notable entre eficiencia y complejidad [8].

Su descripción es la siguiente: Teniéndose un *tour* inicial, se inicia removiendo dos aristas de este *tour*, para luego reconectar los dos caminos resultantes de manera tal que garantiza la

formación de un nuevo *tour* válido; a cada uno de estos cambios se les denomina movimiento 2-opt. Este proceso de reconexión se lleva a cabo únicamente si el nuevo *tour* resulta ser de menor costo; sino, se prueba otra opción de remoción-reconexión. Se repite este ciclo de remoción y reconexión hasta que no se pueden obtener más mejoras 2-opt, momento en el cual el *tour* resultante se denomina "2-optimal" [8].

Ejecutar la heurística 2-opt usualmente resultará en un *tour* con una longitud menos del 5 % por encima de la cota inferior de Held-Karp [8].

Algoritmo 3-opt: Una extensión natural del conocido algoritmo 2-opt que consiste en remover y reconectar tres aristas en lugar de dos. Esta modificación en la estrategia amplía las posibilidades de reconexión, permitiendo una mayor diversidad de formas para reintegrar los caminos en un *tour* válido [8].

Un dato fundamental de este algoritmo, el cual será útil en adelante, es que un movimiento 3-opt puede ser descrito de forma equivalente como a lo más tres movimientos 2-opt sucesivos. Esta equivalencia añade flexibilidad en caso de querer utilizar ambos movimientos, tanto 2-opt como 3-opt, simplificando además la implementación, ya que se puede implementar el algoritmo 3-opt utilizando el algoritmo 2-opt, que es de fácil implementación [8].

Análogamente a como se mencionó en la descripción del algoritmo 2-opt, se define que un *tour* es 3-optimal si no es posible mejorarlo a través de movimientos 3-opt [8].

En base a la definición de 3-optimalidad y 2-optimalidad, es posible relacionar estos dos conceptos según la siguiente regla teórica: si un *tour* es 3-óptimo, entonces también es 2-óptimo [7].

No obstante, aunque se obtienen mejores resultados como consecuencia del aumento en las posibilidades de reconexión, es crucial reconocer que este incremento en la optimalidad conlleva a su vez un aumento en el costo computacional. La complejidad del algoritmo 3-opt se eleva desde los $O(n^2)$ del algoritmo 2-opt a $O(n^3)$. El *trade-off* por esta mayor carga computacional se evidencia al mejorar el rendimiento desde un 5 % por encima de la cota inferior de Held-Karp hasta un 3 % [8].

Algoritmo k-opt: El algoritmo k-opt es una extensión generalizada de las heurísticas 2-opt y 3-opt utilizadas para resolver el TSP. Este enfoque no se limita necesariamente a movimientos 2-opt o 3-opt, sino que puede extenderse a 4-opt, 5-opt y así sucesivamente. Sin embargo, cada aumento en el valor de k conlleva también un incremento significativo en el tiempo de ejecución a $O(n^k)$ para implementaciones *naive*, mientras que las mejoras resultantes del aumento de las posibilidades de reconexión son relativamente pequeñas en comparación con la ganancia al pasar de 2-opt a 3-opt [8].

Es relevante señalar que, al igual que un movimiento 3-opt puede describirse como una secuencia de movimientos 2-opt, un movimiento k-opt también puede representarse como una secuencia de movimientos 2-opt. Sin embargo, existe una excepción significativa que vale la pena mencionar. Este movimiento no representable corresponde a una configuración específica de un movimiento 4-opt conocido como "The Crossing Bridges", un intercambio de cuatro aristas, el cual es realizable con a lo menos dos cambios secuenciales disjuntos de dos aristas,

rompiendo el criterio de secuecialidad.

Este ejemplo ilustra cómo las optimizaciones más allá de 3-opt involucran movimientos más complejos que no se pueden descomponer en secuencias de movimientos más simples. Esto añade una capa adicional de complejidad y desafío computacional al abordar el Problema del Vendedor Viajero [8].

Con respecto a la relación de optimalidad, la conexión entre 2-optimalidad y 3-optimalidad no es exclusiva. Esta relación se extiende a cualquier algoritmo k-opt. La propiedad de que un *tour* que es k-óptimo también es (k-1)-óptimo se mantiene de manera consistente, lo cual resulta fundamental para algoritmos más sofisticados.

En términos de usabilidad de los algoritmos k-opt, se escogen los valores usuales para k de 2 y 3, ya que experimentalmente se ha determinado que valores mayores tienen un altísimo costo, a su vez que un beneficio mínimo e ínfimo. Sin embargo, se han realizado pruebas experimentales con k=4 y k=5, las cuales han demostrado experimentalmente el bajo beneficio y alto costo de estas elecciones [14].

Como otro dato interesante, vale resaltar que los algoritmos de optimización local han demostrado su eficacia al operar incluso sin depender del resultado previo de un algoritmo constructivo, simplemente partiendo de un *tour* inicial aleatorio. En estudios y experimentos, se observa lo anterior, obteniendo resultados sorprendentemente superiores en términos de eficiencia y calidad de la solución al ser aplicados sobre *tour*s aleatorios, con respecto a soluciones dadas por algoritmos constructivos [13].

Algoritmo Lin-Kernighan: Este algoritmo corresponde a una versión significativamente sofisticada, resultado de la búsqueda de algoritmos de optimización local. Se basa en todos los puntos teóricos encontrados en cada uno de los algoritmos de optimización local presentados.

El nacimiento de este algoritmo surge de la pregunta: ¿qué sentido tiene definir el valor de k antes de siquiera empezar a ejecutar el algoritmo? ¿Cómo podemos saber realmente que el mejor cambio a realizar es de k aristas? Y si el mejor cambio a un *tour* es de 6 aristas, o solo 3? Y el mejor cambio siguiente puede ser de solo 2 aristas o quizás 8?

El algoritmo Lin-Kernighan, por lo tanto, presenta una sofisticada variante del enfoque k-opt, donde el valor de k se ajusta dinámicamente iteración tras iteración, de forma ascendente durante la ejecución para determinar la mejor estrategia en cada iteración. Este ajuste variable de k es una característica distintiva que contribuye significativamente a la enorme eficacia de este algoritmo [8].

Aunque el enunciado de este algoritmo pareciera ser simple, como se verá más adelante, se compone de un amplio conjunto de decisiones complejas, las cuales son significativamente flexibles y afectan significativamente al costo y optimalidad del algoritmo. Estas decisiones deben ser tratadas con cuidado, ya que pueden tener importantes efectos tanto en el tiempo de ejecución como en la calidad de la solución.

Como contexto histórico, Lin y Kernighan introdujeron este algoritmo en 1973, y desde entonces, ha demostrado ser un método robusto y complejo, con pocas mejoras evidentes por parte de otros investigadores [8]. Su complejidad temporal es aproximadamente $O(n^{2.2})$ [13], y proporciona resultados que superan en un 2 % la cota inferior de Held-Karp [8].

Esto evidencia claramente tanto el alto nivel de optimalidad con respecto a la cota de Held-Karp visto hasta ahora, con además, uno de los mejores costos computacionales, apenas un poco por encima de 2-opt.

Este algoritmo se presenta como la respuesta en muchos casos, sin ser superado por ninguno de los otros algoritmos desarrollados y coronándose como la solución por antonomasia. Sin embargo, su costo radica en su complejísima implementación.

El algoritmo, al ser de optimización local, se centra en el enunciado de TSP, que consiste en definir el conjunto de aristas que minimiza el costo y mantiene el invariante de *tour*. En ese sentido, se trabaja directamente con énfasis en las aristas.

La descripción detallada de este algoritmo se encuentra en el artículo original [13] y se puede resumir de la siguiente manera:

1. Sea E el conjunto de aristas del grafo, luego, se genera una solución inicial aleatoria T siendo T un conjunto de aristas que determina un *tour*.
2. (a) Establece $i = 1$.
   (b) Selecciona las aristas $x_i \in T$ e $y_i \in E - T$ como el par de aristas más fuera de lugar en el paso i. Esto generalmente significa que x_i e y_i se eligen para maximizar la mejora cuando $x_1, \ldots, x_i$ se intercambian con $y_1, \ldots, y_i$. x_i se elige de $T - \{x_1, \ldots, x_{i-1}\}$ e y_i de $E - T - \{y_1, \ldots, y_{i-1}\}$.
   (c) Si parece que no se puede obtener más ganancia, de acuerdo con una regla de detención apropiada, ve al Paso 3; de lo contrario, establece $i = i + 1$ y regresa al Paso 2(b).
3. Se encuentra la mejor mejora para $i = k$, intercambia $x_1, \ldots, x_k$ con $y_1, \ldots, y_k$, haciéndose un movimiento k-opt, para obtener un nuevo *tour* T, y vuelve al Paso 2; si no se encuentra ninguna mejora, ve al Paso 4.
4. Repite desde el Paso 1 si así se desea.

Al leer la descripción anterior, se evidencia que hay múltiples decisiones que tomar. Veamos cada una de ellas en base a extractos de texto de la descripción.

- "selecciona x_i e y_i como el par más fuera de lugar": Se requiere establecer un *criterio de elección de x e y* que, dado un conjunto de aristas, determine cuál está más fuera de lugar. Este criterio de selección es crucial para la efectividad del algoritmo, ya que influirá en las mejoras que se puedan lograr en cada iteración.
- "Si parece que no se puede obtener mas ganancia": Se requiere entonces definir una*función de ganancia* que, para una serie de movimientos, determine si puede o no generarse más ganancia con algún movimiento posterior.

- "x_i e y_i se eligen para maximizar la mejora": Requiere definir una *función de mejora* que determine si una elección específica de x_i e y_i mejora la solución. La funcionalidad de mejora debe evaluar el impacto de cambiar las aristas x_i e y_i, seleccionando aquellas que maximicen positivamente dicho impacto en la solución del TSP. La efectividad del algoritmo Lin-Kernighan depende en gran medida de cómo se diseñen y optimicen estas funciones, ya que influyen en las decisiones cruciales durante la ejecución del algoritmo.

- "De acuerdo con una regla de detención apropiada": Requiere definir una *regla de detención apropiada*. La regla de detención es crucial para determinar cuándo finaliza el algoritmo Lin-Kernighan. Puede basarse en criterios específicos, como el número de iteraciones, la mejora en la solución o cualquier otro indicador relevante. El diseño adecuado de esta regla impacta directamente en la eficiencia y efectividad del algoritmo. Es fundamental seleccionar criterios que permitan un equilibrio óptimo entre la calidad de la solución obtenida y el tiempo de ejecución del algoritmo.

- "Haciéndose un movimiento k-opt" y "Para obtener un nuevo *tour*": Se requiere definir un *criterio de factibilidad*, el cual determine que las aristas escogidas para el movimiento k-opt, al ser cambiadas, conformen un *tour* factible. Este criterio es esencial para garantizar que el nuevo *tour* resultante sea válido y cumpla con las restricciones del problema del TSP.

- "Si no encuentra ninguna mejora": Se requiere definir el *criterio de finalización*, que determine cuándo el algoritmo debe dejar de buscar mejoras al no encontrar ninguna mejora adicional. Este criterio es esencial para controlar la ejecución del algoritmo y evitar que continúe indefinidamente en busca de mejoras que no sean significativas.

Frente a esto, Lin y Kernighan definen las siguientes elecciones:

Criterio de elección de x e y: Se establece el llamado Criterio de disjuntividad: El conjunto $X = \{x_i\}_i$ debe ser disjunto al conjunto $Y = \{yi\}_i$ en todo momento, es decir, no debemos agregar una arista que ya fue eliminada o viceversa.

Adicionalmente, la elección de yi debe ser uno de los 5 vecinos más cercanos a un extremo de x_i buscando maximizar la ganancia dado un x_i ya escogido. Este criterio asegura una selección coherente y efectiva de x e y para el movimiento k-opt, aunque quizás demasiado ingenuo, como se verá más adelante.

Función de ganancia: Esta función se calcula para cada iteración después de la elección de una arista x_i a extraer y una arista y_i a extraer. Se define la ganancia de esta acción como g_i, tal que $g_i = c(x_i) - c(y_i)$, siendo c la función que a cada arista retorna su costo.

Función de mejora: Se define como la suma acumulada de ganancias a través de las iteraciones del algoritmo. Para la iteración j-ésima, se tiene una mejora de G_j con $G_j = \Sigma_{i=1}^{j} g_i < 0$. La función de mejora cuantifica la mejora acumulativa lograda durante las iteraciones del algoritmo.

Regla de detención apropiada: Se define el llamado "Criterio de Ganancia Positiva", el cual determina que la detención del proceso de determinar k para realizar un movimiento

k-opt se detiene en $k = r$ si $G_r < 0$. Es decir, se detiene cuando la suma acumulada de ganancias se vuelve negativa. Vale la pena mencionar que la ventaja de esta elección es que permite potencialmente la presencia de ganancias negativas en algunas iteraciones, sin detenerse frente a ellas en caso de tener una ganancia total positiva. Esto permite seguir explorando la solución en busca de probables beneficios más adelante que compensen la pérdida inmediata, lo cual experimentalmente se demostró efectivo.

Criterio de factibilidad: Se define el llamado "Criterio de Intercambio Secuencial", el cual establece que la arista a remover x_i y la arista a ser agregada y_i deben compartir un vértice en común. En particular, si definimos $x_i = (t_{2i-1}, t_{2i})$ como la arista a quitar en la iteración i-ésima del algoritmo, siendo t_{2i-1} y t_{2i} dos vértices unidos por esta arista, entonces y_i necesariamente debe ser de la forma $y_i = (t_{2i}, t_{2i+1})$ recordando que t_i es la arista escogida en la i-ésima iteración. Esto permite describir los movimientos k-opt como secuencias de movimientos 2-opt.

Vale mencionar que el movimiento 4-opt "Crossing Bridges" no es representable como una secuencia de movimientos 2-opt, pero sin embargo es útil, en cuanto hay *tours* que pueden ser optimizables únicamente utilizando este movimiento. Para esto, es necesario definir su utilización al encontrarse en un óptimo local de Lin-Kernighan, evitándolo a través de los pasos iterativos del mismo.

Adicionalmente a este criterio, se define el "Criterio de Factibilidad", el cual indica que la arista a extraer $x = (t_{2i-1}, t_{2i})$ debe ser escogida de tal manera que, si unimos t_{2i} con t_1 o sea el primer vértice donde se inicializa la ejecución el algoritmo, se obtenga un *tour*. Es decir, tras cada iteración, sea posible formar un *tour* en ese momento si así fuera deseado al realizar una conexión.

Proceso de finalización: Al elegir un primer vértice sobre el cual se quiere comenzar la ejecución del algoritmo Lin-Kernighan, es claro que tenemos dos opciones para elegir la primera arista a extraer, x_1, cada una de las dos aristas que conectan este vértice al *tour*. Esta variedad de opciones también ocurre para la primera arista a agregar, y_1, y también para x_2 y para y_2.

Luego, tras haberse determinado el término del proceso iterativo del algoritmo Lin-Kernighan, se prueba partiendo desde esta otra opción de la primera arista.

Una vez probada esta opción y verificado que su costo fuera mayor o menor al obtenido con la opción original, se da por finalizado el algoritmo de Lin-Kernighan, obteniéndose un óptimo, ya sea local o global. Ahora es posible volver a ejecutar el algoritmo desde un *tour* inicial distinto.

Este algoritmo cuenta con múltiples casos particulares y consideraciones específicas a tener en cuenta, sobre todo con casos en que puede romperse temporalmente el criterio de intercambio secuencial durante algunas iteraciones, para luego recuperarlo, sacrificándolo durante algunas iteraciones para probar nuevos caminos. Sin embargo, estas especificaciones corresponden a aspectos técnicos relevantes para la implementación de la heurística con el objetivo de optimizar el tiempo de ejecución en lugar de lograr el entendimiento del mismo

algoritmo, y se consideran demasiado específicas y sin mayor relevancia para esta memoria. Dichas decisiones pueden ser leídas detalladamente en el artículo original [13].

A continuación se mencionan algunas de estas decisiones para dar un ejemplo del tipo al que corresponden:

- Si una arista está presente en más de dos óptimos locales obtenidos, se prohíbe su extracción del *tour* al ser probablemente parte del óptimo. Esto permite evitar alejarse del óptimo, convergiendo probablemente con más rapidez.
- Al finalizar el proceso de optimización mediante movimientos secuenciales k-opt y antes de revisar los caminos alternativos de x_1, x_2, y_1, y_2 se compara el *tour* obtenido con el conjunto de óptimos locales. En caso de que coincida con un óptimo local, no se prueban los caminos alternativos, ahorrando así tiempo de ejecución.

Así como estas decisiones, hay múltiples, las cuales, como se puede notar, no generan un mayor entendimiento del algoritmo, sino que permiten reducir su costo computacional, exprimiendo al máximo el rendimiento. Aunque cada una de estas múltiples decisiones adicionales permite reducir el costo del algoritmo, también complejizan la implementación del mismo.

El desarrollo de este complejo algoritmo, con las decisiones particulares de Lin y Kernighan, obtiene resultados que, hasta el día de hoy, 50 años después, siguen siendo significativos. Hoy en día, con los computadores actuales, se puede obtener la solución a problemas de menos de 100 ciudades en apenas un instante, décimas de segundo, incluso centésimas de segundo. Para problemas de 50 ciudades, la probabilidad de obtener la solución óptima en una sola ejecución es cercana al 100 %, mientras que para problemas de alrededor de 100 ciudades, la probabilidad se reduce a entre un 20 % y 30 %. Sin embargo, dado un conjunto de ejecuciones, el óptimo puede ser encontrado en uno de ellos con casi un 100 % de confianza [7].

Al momento de ser publicado el algoritmo de Lin-Kernighan en el año 1973, se obtuvo una solución a un problema de 318 vértices, logrando a través de este algoritmo una nueva cota superior. Vale la pena señalar que ahora que se conoce el óptimo de este problema, su cota superior estuvo un 1.3 % por encima del óptimo real [7].

Como se puede observar, el algoritmo heurístico de Lin-Kernighan conlleva un amplio conjunto de múltiples decisiones que, a su vez, son heurísticas. Esta complejidad implica una desventaja al complicar su implementación, haciéndola muy específica. Sin embargo, también es una ventaja, ya que cada una de las decisiones puede ser manipulada individualmente, dando lugar a diversos comportamientos y distintas versiones de LK definidas por el conjunto de elecciones que se realice.

Respecto a cómo un conjunto de decisiones puede definir distintas versiones del algoritmo Lin-Kernighan, un conjunto de esas elecciones, el cual ha tenido resultados increíbles, ha sido el propuesto por Keld Helsgaun en sus diversos artículos. En cada uno de ellos, Helsgaun refina o modifica las elecciones, permitiendo obtener resultados altamente prometedores y convirtiéndose en un estándar indiscutible en el estudio de heurísticas para TSP.

Como conclusión a la presentación del algoritmo Lin-Kernighan, se obtiene que, a lo largo del tiempo, existe un consenso general de que el algoritmo Lin-Kernighan y sus modificaciones son los más eficientes en términos del equilibrio entre costo computacional, tiempo y grado de optimalidad. Este reconocimiento se remonta a 1989 [7], se confirma en el año 2003 [8], y perdura hasta el día de hoy con versiones mejoradas presentadas en el 2023 [15].

Queda entonces profundizar aún más en esta heurística, especialmente en la prometedora versión de Lin-Kernighan con el conjunto de decisiones definido por Keld Helsgaun.

Algoritmo Lin-Kernighan-Helsgaun: Aunque los resultados obtenidos por el algoritmo Lin-Kernighan eran razonablemente efectivos, al menos para problemas de alrededor de 50 ciudades, la probabilidad de obtener el óptimo tras cada ejecución disminuía rápidamente al acercarse a las 100 ciudades, quedando entre un 20 % y 30 %.

Adicionalmente, no era capaz de obtener el óptimo para el problema de 318 vértices en el que se probaba, mientras que los casos interesantes para Synopsys están en el orden de los miles de *scan flops*.

Un análisis crítico a sus heurísticas desarrollado por Keld Helsgaun le permitió mejorar los resultados.

En particular y fundamentalmente, cuestiona el criterio de elegir la arista a agregar como aquella que conecta a un vértice entre a lo sumo los cinco vecinos más cercanos. Este cuestionamiento se justifica, por ejemplo, en un caso emblemático de 532 ciudades, donde una de las aristas que constituye el *tour* óptimo corresponde al vigesimosegundo vecino más cercano. En consecuencia, dado el criterio de Lin-Kernighan de solo agregar aristas de los cinco vecinos más cercanos, jamás podría encontrar el óptimo en este problema.

El hecho de simplemente aumentar la revisión de vecinos más cercanos a 22 no solo es altamente ineficiente en tiempo, sino que tampoco es la solución, ya que se pueden diseñar casos, e incluso obtenerlos naturalmente, en los cuales, por ejemplo, alguna arista del *tour* óptimo sea más lejana que la vigesimosegunda, como la vigesimocuarta o, en general, cualquier orden mayor.

Esto pone en evidencia la necesidad de modificar el criterio de Lin-Kernighan para abordar este problema y aquí radica probablemente la mayor limitación del algoritmo.

Como respuesta a este problema, Keld Helsgaun propone una solución muy interesante.

Define como *1-tree* el grafo construido como un árbol de expansión mínima del grafo, sumado a una arista adicional que genere un ciclo en el grafo.

Notando que la longitud de un *minimum 1-tree* corresponde a una cota inferior de la solución óptima, encuentra una forma de construir un *minimum 1-tree* tal que, además de ser una cota inferior, sea la más alta cota inferior, con mucha probabilidad, acercándose así al óptimo real.

En sus resultados experimentales, obtiene que, para un conjunto de pruebas ampliamente reconocido para el cual se conoce el óptimo exacto, el error promedio de la cota inferior con respecto al óptimo real es de alrededor del 1.1 %.

Luego, define una nueva métrica de distancia para aristas, basada en su cercanía a formar parte del *1-tree*. A esta nueva distancia la denomina alfa-distancia y la utiliza en reemplazo de la distancia al vecino más cercano como la medida de distancia en el espacio.

Con esta corrección, no solo logra encontrar el óptimo para el caso emblemático presentado de 532 vértices, sino que además, en 98 de 100 ejecuciones de su algoritmo, alcanza el óptimo.

No solo eso, sino que además este cambio en la forma de definir la cercanía permite determinar que si tomamos en cuenta únicamente los 5 vecinos más alfa-cercanos, se mejora la calidad y efectividad del algoritmo para todos los casos antes probados en Lin-Kernighan. Se logra obtener el óptimo el 100 % de las veces para casos de 100 vértices, cuando antes, con Lin-Kernighan original, se obtenía entre un 20 % y 30 %. Pensando en las máximas capacidades, este algoritmo además logra encontrar una solución un 0.02 % por encima del óptimo real para un caso de 85900 vértices con dos semanas de computación en un procesador de 300MHz, lo cual es realmente una hazaña y una mejora increíble con respecto a la versión original de Lin-Kernighan. Incluso, Helsgaun llega a recomendar este algoritmo para casos de hasta un máximo de 100.000 vértices. Esto coincide justamente con los intereses técnicos de Synopsys [7].

La complejidad computacional se mantiene inalterada con respecto a Lin-Kernighan, es decir, aproximadamente $O(n^{2.2})$. Esto implica que el costo sigue siendo del mismo orden que el algoritmo original de Lin-Kernighan.

El costo frente a este beneficio, que implica mantener el rendimiento a la vez que mejora de manera notable la solución, es la aún más compleja implementación con respecto al algoritmo original de Lin-Kernighan.

Este algoritmo ha sido calificado como una "mejora considerable del algoritmo original (Algoritmo Lin-Kernighan)" [7]. Como dato adicional, cabe destacar que ha logrado obtener soluciones óptimas para prácticamente todos los casos de TSPLib, una librería estándar de casos de prueba para el TSP, que será explorada en mayor profundidad más adelante en este trabajo. Aunque los resultados datan de 1998, siguen siendo vigentes y son reconocidos hasta el día de hoy como el estado del arte en la resolución del TSP [15].

2.5. Descripción concreta del Algoritmo Lin-Kernighan-Helsgaun

Habiéndose introducido el algoritmo de Lin-Kernighan con las mejoras de Keld Helsgaun y al identificar su promisorio desempeño, se procede hacia una descripción más detallada de este algoritmo con el fin de describir cada uno de sus pasos. Esta descripción se hace en base al artículo original [7]

Dado que el algoritmo de Helsgaun es una versión del algoritmo Lin-Kernighan, basada en la definición de un conjunto diferente de decisiones, procedemos a establecer las diferencias en las elecciones entre ambos.

2.5.1. Criterio de elección de aristas

Como se explicó en la introducción de este algoritmo, se modifica la noción de "vecinos más cercanos" a "vecinos más alfa-cercanos". Esta modificación se justifica por las siguientes razones:

- Usualmente, el *minimum 1-tree* contiene alrededor del 70 % al 80 % de las aristas del *tour* óptimo. Por lo tanto, la distancia se mide como la diferencia entre las aristas del *tour* actual y las aristas del *minimum 1-tree*. Se otorga una mejor calificación a las aristas que se clasifican como más cercanas al *minimum 1-tree*. A continuación, se define de manera más precisa esta noción de distancia.
- Matemáticamente si T es un *minimum 1-tree* de largo $L(T)$ y $T^+(i,j)$ el *minimum 1-tree* que obligatoriamente tiene la arista (i,j), se define la función de distancia llamada alfa-distancia sobre las aristas de la forma $\alpha(i,j) = L(T^+(i,j)) - L(T)$. Es claro que si la arista (i,j) pertenece al *minimum 1-tree*, entonces la alfa-distancia es 0.

En cuanto a la elección de x la arista a extraer, se define la siguiente regla adicional específica: x_1, la primera arista a extraer, no puede pertenecer al *minimum 1-tree*.

Respecto a las demás decisiones de Lin-Kernighan, como la función de ganancia, la regla de detención apropiada, el criterio de fuera de lugar, el criterio de factibilidad y el criterio de no seguir revisando, se mantienen practicamente idénticas.

2.5.2. Proceso de Ascent

Es importante destacar que cualquier *tour* es, a su vez, un *1-tree*. Por lo tanto, la longitud de un *minimum 1-tree* constituye una cota inferior de la longitud del *tour* óptimo, y se utiliza como tal.

Adicionalmente, según [16], si aumentamos en un valor, digamos π el costo de todas las aristas incidentes a un mismo nodo, el *tour* óptimo se compone de las mismas aristas. Sin embargo el *minimum 1-tree* cambia. Aún más, podemos redefinir el costo de todas las aristas desde un costo $c(i,j)$ a un nuevo costo $d(i,j)$ tal que $d(i,j) = c_(i,j) + \pi_i + \pi_j$ generando un nuevo *minimum 1-tree*, llamémoslo T_π.

Según el mismo trabajo [16], la longitud del *minimum 1-tree* T_π sigue siendo cota inferior del *tour* óptimo. Dado que $L(T_\pi) = L(T_\pi) - 2\Sigma\pi_i$, podemos maximizar esta función, obteniendo así una máxima cota inferior. Esta cota resulta muy útil para medir el rendimiento de las heurísticas para resolver el Problema del Vendedor Viajero, ya que esta cota inferior suele estar a un 1.1 % del óptimo real.

El método utilizado para maximizar la cota inferior dada por L_π es mediante el método de *Subgradient Optimization*, como se describe en el artículo [16]. Este método es un enfoque numérico iterativo que ajusta los valores de π paso a paso en cada iteración. Los valores π se denominan penalizaciones, y el proceso de obtener las penalizaciones óptimas que generan la mayor cota inferior se denomina Ascent.

Esta nueva definición de distancia resulta mucho más efectiva para encontrar óptimos en comparación con la anterior distancia de vecino más cercano definida por Lin y Kernighan en su algoritmo original. De hecho, según [7], "En todos los problemas, el algoritmo fue capaz de encontrar el *tour* óptimo usando como aristas candidatas solamente las cinco aristas con menos alfa-distancia; aún más, la mayoría de los problemas pudo ser incluso resuelto cuando las aristas candidatas por nodo fueron reducidas a las 4 más alfa-cercanas". El resultado es que se revisan menos aristas candidatas y, al mismo tiempo, se obtiene una mejor calidad de solución, esto es una mejora absoluta.

Tras examinar cada uno de estos algoritmos de optimización iterativa y su culminación en el algoritmo de Lin-Kernighan, seguido por su mejora por Helsgaun, parece ser evidente su supremacía. Sin embargo, es importante señalar que estos algoritmos fueron propuestos exclusivamente para el caso TSP simétrico, el cual no corresponde a nuestro problema de *Reordering*, que es una instancia de TSP asimétrico.

Aunque en la literatura no se ha discutido la realización de una versión alternativa de los algoritmos basados en movimientos k-opt para el caso asimétrico, se ha observado que estos algoritmos son adaptables, al menos para 2-opt y 3-opt, de una manera no completamente documentada. Sin embargo, estas adaptaciones resultan en soluciones excesivamente complejas. Estas soluciones se vuelven difíciles de entender y, además, agregan múltiples costos lineales en cada iteración, ralentizando enormemente el algoritmo sin mejorar su rendimiento. Un ejemplo de esto es Synopsys, que, basándose en la idea de algoritmos basados en movimientos k-opt, ha intentado adaptarlos para funcionar directamente en casos asimétricos. Esto ha llevado a soluciones excesivamente complicadas, difíciles de extender y mejorar, desconectándolas del desarrollo académico al desaprovechar las ventajas de la investigación de heurísticas en el caso simétrico.

En lugar de adaptar los algoritmos basados en movimientos k-opt para operar directamente sobre casos asimétricos, una buena idea podría ser transformar el problema asimétrico a uno simétrico equivalente. De esta manera, se podría aprovechar el potencial de estos prometedores algoritmos de optimización local diseñados originalmente para el caso simétrico.

2.6. Conversión desde TSP asimétrico a un equivalente simétrico

En el artículo [17], Jonker y Volgenant muestran la forma de convertir un Problema del Vendedor Viajero asimétrico de n nodos a uno equivalente simétrico de $2n$ nodos, cuando los costos se obtienen a partir de una matriz de costos.

La descripción matemática de esta conversión es la siguiente:

Sea $C = (c_{i,j})$ la matriz de costos que retorna el costo de la arista (i, j).

$c'_{(n+i,j)} = c'_{(j,n+i)} = c_{(i,j)}, \quad$ para $i = 1, 2, ..., n,\ j = 1, 2, ..., n,$ y $i \neq j$

$c'_{(n+i,i)} = c'_{(i,n+i)} = -M$ para $i = 1, 2, ..., n$

$c'_{(i,j)} = M$

La desventaja es obvia: al depender la complejidad del algoritmo directamente del número de vértices, una duplicación de los mismos puede parecer una desventaja evidente, la cual podría ser descartada de inmediato. Además, el tiempo empleado en el preprocesamiento, es decir, la conversión desde un caso asimétrico a uno simétrico, el cálculo de la nueva matriz de costos y el desarrollo de una funcionalidad adicional que determine qué vértices pertenecen al grafo y cuáles son virtuales (usados para la conversión), pueden parecer nuevamente excesivamente costosos.

Sin embargo, los algoritmos Lin-Kernighan y Lin-Kernighan-Helsgaun resultan altamente prometedores. Sus niveles de optimalidad y tiempo de ejecución son enormemente prometedores y, en primera instancia, pareciera que compensan todos los gastos de preprocesamiento.

Capítulo 3

Solución

Como solución, se decide implementar el algoritmo Lin-Kernighan con el conjunto de decisiones de Helsgaun. Este algoritmo es altamente complejo de implementar, con una descripción dada en el artículo [7], un artículo que abarca más de 70 páginas.

Para presentar su implementación, se describen los aspectos técnicos desarrollados para la implementación de cada una de las funcionalidades descritas en el artículo original. Además, se ahonda aún más en el funcionamiento del algoritmo.

El artículo original proporciona una descripción en amplio detalle, la cual no será tratada en su totalidad en este capítulo. En su lugar, se abordarán sus componentes más fundamentales, dejando al lector el artículo guía [7] como referencia completa.

Sin embargo, antes de decidir cómo realizar la implementación, es necesario conocer que características debe cumplir la implementación. Para ello se presenta un conjunto de características que debe cumplir la implementación deseada.

3.1. Estándares del algoritmo a implementar

El estándar fundamental corresponde al tipo de entrada, es decir, cómo se describen las aristas en los archivos de entrada de TSP. A lo largo de la historia del desarrollo de heurísticas para TSP, la biblioteca TSPLib ha sido indiscutiblemente el estándar, siendo probados todos los algoritmos sobre ella.

Concretamente, la biblioteca TSPLib corresponde a un conjunto de casos de prueba para diversos algoritmos de optimización combinatorial en grafos. Esto incluye el Problema del Vendedor Viajero simétrico, el Problema del Vendedor Viajero asimétrico, el Problema de Enrutamiento de Vehículos, el Problema del Orden Secuencial y el Problema de determinar si un grafo contiene un ciclo hamiltoniano. Cada uno de estos problemas se define sobre un grafo completo cuyos costos son enteros.

En particular, los casos de prueba para los problemas simétrico y asimétrico se componen de un conjunto de inputs del Problema del Vendedor Viajero para los cuales se conoce la longitud del *tour* óptimo y el orden de los vértices en su *tour* óptimo.

Profundizando en términos técnicos específicos de esta biblioteca para el Problema del Vendedor Viajero, el costo de cada arista se puede definir de varias formas estandarizadas diferentes. Por ejemplo, como una matriz explícita de costos triangular inferior o superior para los casos simétricos, una matriz explícita completa para los casos asimétricos, o un conjunto de coordenadas en el plano bidimensional para cada uno de los vértices que conforman el problema, calculando distancias entre ellos según, por ejemplo, distancia euclidiana, de Manhattan o geográfica, entre otros.

Dado que el único estándar para el caso asimétrico corresponde a la matriz explícita, se incorpora este estándar de archivo de entrada. Además, el algoritmo es válido para versiones simétricas, por lo que se implementa algunos de los estándares de TSP simétrico. En particular, se implementa el caso euclidiano, ya que los archivos de prueba originales de Synopsys corresponden a un conjunto de coordenadas en el espacio, similar a este estándar. Esto facilita el desarrollo futuro de nuevas funcionalidades directamente sobre archivos de coordenadas que pueda definir Synopsys. Por lo tanto, solo estos dos tipos de casos de TSPLib son relevantes para esta memoria.

Para ilustrar la forma de este archivo, se presenta a continuación un extracto de la porción inicial del caso "a280.tsp", un caso de prueba de 280 vértices que corresponde a una instancia simétrica euclidiana de TSPLib:

Código 3.1: Extracto de archivo euclídeo simétrico de TSPLib

```
NAME: a280
TYPE: TSP
COMMENT: drilling problem (Ludwig)
DIMENSION: 280
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 288 149
2 288 129
3 270 133
4 256 141
5 256 157
6 246 157
7 236 169
...
```

Se observa que se define el nombre de la instancia, luego un breve comentario sobre ella, usualmente sobre su origen. Luego se especifica su tipo, en este caso, TSP, ya que corresponde a TSP simétrico. A continuación, se indica el número de vértices, que en este caso es 280. Se especifica que la función de costos utilizada para esta instancia es la norma euclidiana en dos dimensiones ("EDGE_WEIGHT_TYPE: EUC_2D"). Finalmente, se presenta una lista con las coordenadas en dos dimensiones de cada uno de los vértices que componen el caso de prueba en la sección "NODE_COORD_SECTION".

Por su parte, se muestra un extracto del inicio de un archivo de pruebas de TSP asimé-

trico, el caso “br17.atsp”:

Código 3.2: Extracto de archivo asimétrico de TSPLib

```
NAME: br17
TYPE: ATSP
COMMENT: 17 city problem (Repetto)
DIMENSION: 17
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
9999  3  5  48  48  8  8  5  5  3  3  0  3  5  8  8  5
3 9999  3  48  48  8  8  5  5  0  0  3  0  3  8  8  5
...
```

Al igual que para el caso euclidiano, se muestra el nombre de la instancia, luego su tipo (“ATSP” que corresponde a TSP asimétrico), un comentario sobre el caso de prueba, el número de vértices del caso. Luego se indica que la función de costos es explícita, es decir, se obtienen directamente del archivo de entrada. Finalmente, se presenta la matriz de costos completa. En este caso, solo se muestran las primeras dos columnas para ilustrar el archivo de entrada.

Con miras a desarrollar un algoritmo de Synopsys compatible más compatible con aquellos conocidos en la literatura, el algoritmo implementado lee archivos de entrada en formato TSPLib, tanto para TSP euclidiano como asimétrico con matriz de costos explícita.

Habiendo establecido el estándar a seguir, procedemos ahora a describir la implementación del algoritmo de Lin-Kernighan Helsgaun (LKH).

3.2. Implementación del algoritmo

En esta sección, abordaremos los aspectos técnicos enfrentados y desarrollados durante la implementación del algoritmo LKH. La implementación se llevó a cabo en el lenguaje de programación C, basándose en los códigos y descripciones proporcionados por Helsgaun en su artículo [7].

La implementación total consta de poco menos de 3000 líneas de código. Aunque existen múltiples funcionalidades que ocupan más de 200 líneas, que resultan repetitivas, como la realización de movimientos 5-opt, los cuales se pueden llevar a cabo de 52 formas distintas.

También hay funciones aparentemente simples pero que involucran múltiples líneas de código, como la lectura de la entrada, que utiliza más de 300 líneas.

3.2.1. Implementación de vértices

La implementación de los vértices es fundamental, ya que define la forma en que se ejecutarán las funcionalidades. Helsgaun señala que el mayor cuello de botella radica en la búsqueda de movimientos k-opt a realizar y la funcionalidad de llevar a cabo el movimiento determinado. Con base en esto, se implementa la estructura de vértice de manera que permita ejecutar estas dos operaciones de manera eficiente. Siguiendo las directrices de Helsgaun, se busca implementar una estructura que brinde información sobre el vértice predecesor y sucesor, y además, tenga la capacidad de determinar, dados dos vértices, si se encuentra entre ellos en el *tour* actual.

Para lograr esto, se ha creado una estructura de datos "Vertice", que consta de varios atributos, incluyendo un puntero al vértice anterior y posterior en el *tour*. En relación con la funcionalidad de determinar si un vértice se encuentra entre otros dos, se ha implementado un atributo denominado "Orden", que representa la posición relativa del nodo en el *tour*. La comparación de los atributos Orden de cada uno de los tres vértices permite determinar eficientemente si un vértice se encuentra entre otros dos vértices.

Dado el uso transversal de la estructura "Vertice" a lo largo del algoritmo, es esencial que también cuente con los siguientes atributos:

- Dos enteros, X e Y, que corresponden a su ubicación en el espacio, utilizados especialmente en el caso euclidiano.
- Un puntero a la fila de la matriz de costos que corresponde a ese vértice. Este puntero es útil para el caso de matriz completa explícita, que se utiliza en el caso asimétrico.
- Un arreglo con estructuras de aristas salientes de este vértice. Este arreglo contiene las cinco aristas más cercanas según el criterio alfa, como recomienda el artículo de Helsgaun.
- Índice: Un valor entero que identifica de manera única cada vértice en el grafo. Esta identificación permite la construcción de cadenas de enteros que definen el orden de los vértices en el *tour* solución, facilitando así la generación de archivos con el orden del *tour* óptimo.
- Un entero con el valor de π asociado a este nodo, utilizado durante el proceso de Ascent para obtener el *minimum 1-tree* de cota inferior maximal.

Cada uno de estos atributos adicionales enumerados, aunque no son fundamentales para la estructura básica del algoritmo, contribuyen a la eficiencia y funcionalidad de la implementación. Por ejemplo, un entero que almacena el grado del vértice durante el proceso de construcción del *minimum 1-tree*. Este atributo permite medir la diferencia entre el *minimum 1-tree* obtenido y el *tour* óptimo, mediante la sumatoria de grados superiores a dos.

Existen otros atributos más específicos que no son esenciales para la implementación, y su utilidad dependerá de enfoques particulares que se le quiera dar al algoritmo.

Con respecto a las aristas, se implementan como estructuras simples con un booleano que indica el sentido en que se recorre la arista, un puntero al vértice de origen y otro al vértice de destino, un entero con el costo asociado a la arista y otro entero con su valor de alfa-distancia.

Estos atributos específicos contribuyen a la adaptabilidad del algoritmo a distintos contextos y permiten su ajuste según las necesidades particulares del problema.

3.2.2. Implementación de conversión ATSP-TSP

En el caso asimétrico, donde se recibe una matriz explícita, se lleva a cabo la duplicación de ambas dimensiones de la matriz bidimensional. Posteriormente, se obtiene la nueva matriz de costos siguiendo las reglas establecidas por Jonker y Volgenant [17], las cuales ya han sido expuestas.

Es relevante destacar que el tiempo necesario para el cálculo de la matriz de costos se denomina tiempo de preprocesamiento. Este tiempo será de interés medir en el futuro para determinar el costo asociado a esta conversión.

Se ha definido, además, una función sencilla que identifica si un vértice es virtual o no virtual. Esto evita que se contabilice en la función de ganancia en cada iteración, así como en la determinación del costo del árbol de expansión mínimo y en la identificación del mejor movimiento a realizar.

3.2.3. Lectura del archivo de entrada

La elección de leer el formato de TSPLib para el archivo de entrada se fundamenta en permitir que Synopsys se ajuste al estándar académico, aprovechando los beneficios asociados. La implementación se realiza en dos formas de input. En primer lugar, se opta por un input euclidiano. Esta elección presenta dos ventajas significativas. Por un lado, permite medir el rendimiento del algoritmo para casos simétricos en comparación con los casos de TSPLib, proporcionando una mayor cantidad de resultados para evaluar el comportamiento del algoritmo. Estos resultados son extensibles a ATSP, ya que la resolución de un caso ATSP ocurre en el mismo proceso que la resolución de un caso simétrico, dado que la conversión de su versión asimétrica a su versión simétrica equivalente ya ha sido explicada.

Adicionalmente, el uso de coordenadas es de interés para Synopsys, ya que se busca trabajar directamente con las coordenadas de los *scan flops* en lugar de la matriz de costos explícita. Esta matriz se vuelve ineficiente en tiempo de preprocesamiento y espacio para casos de mayor tamaño en los que Synopsys operará en el futuro.

3.2.4. Variables de ejecución del algoritmo

A lo largo de la ejecución del programa, existen variables que se utilizan de manera transversal en todas sus funcionalidades, tales como:

- Un entero que almacena el largo óptimo del *tour* dado como entrada en caso de conocerse. Este valor resulta especialmente útil en los casos de prueba de TSPLib, permitiendo la comparación de la solución desarrollada con el óptimo en cada cadena y cada ejecución.
- Un arreglo que contiene todos los vértices del problema, creados inicialmente.
- Un entero que almacena el valor de la cota inferior obtenida durante el proceso de Ascent.
- Una tabla de Hash en la que se guardan los *tours* óptimos locales obtenidos en múltiples ejecuciones de una cadena. Esto facilita la comparación de un *tour* encontrado con previos óptimos locales, con el fin de evitar el costo computacional innecesario del proceso de finalización en caso de coincidir con un óptimo local previamente obtenido. La clave de la tabla corresponde a la longitud del *tour* óptimo local, y el valor corresponde a un conjunto de vértices conectados entre sí a través de sus punteros siguiente y anterior.
- La ruta de sistema donde se desea crear el archivo de salida que contiene el orden del *tour* óptimo, representado por el orden de aparición de los índices de cada vértice.
- Variables relacionadas con la lectura del archivo de entrada, como el tipo de representación de las aristas, la naturaleza del problema (simétrico o asimétrico), y la función de costos. La flexibilidad en estas variables permite que Synopsys pueda implementar fácilmente nuevas funciones de costos en el futuro, como por ejemplo, una función de costos Manhattan que opere directamente sobre casos asimétricos dados por las posiciones de entrada y salida del *scan flop*.

Estas son solo algunas de las variables transversales, habiendo también otras más específicas.

3.2.5. Implementación del Proceso de Ascent

En la implementación del proceso de Ascent, se ha definido como parámetro del programa el número de vecinos más alfa-cercanos a considerar. El proceso se desarrolla de la siguiente manera:

1. Cálculo del *Minimum Spanning Tree* (MST): Se inicia calculando el *minimum spanning tree*, y se elige el algoritmo de Prim por su eficiencia, especialmente en grafos densos como aquellos representados por grafos completos.
2. Cierre del Árbol Formando un Ciclo: Se procede a cerrar el árbol, formando un ciclo. Esto se logra agregando una arista adicional a una de las hojas del árbol y conectándola a su vecino más cercano.
3. Aplicación del Algoritmo de *Subgradient Optimization*: Se utiliza el algoritmo de *subgradient optimization* con un paso decreciente. En cada paso, se obtienen valores de π que generan *minimum 1-trees*. Estos árboles definen cotas inferiores cada vez mayores.

Este proceso de Ascent busca mejorar gradualmente la cota inferior a medida que se ajustan los valores de π. La elección del número de vecinos más alfa-cercanos y la implementación de algoritmos eficientes para el cálculo del MST contribuyen a la efectividad del proceso y a la capacidad del algoritmo para encontrar soluciones óptimas.

3.2.6. Implementación de movimientos k-opt

Como se introdujo previamente, los movimientos k-opt con $k > 2$ pueden describirse como una secuencia de movimientos 2-opt. Por ejemplo, un movimiento 3-opt o 4-opt se puede describir como una secuencia de, a lo sumo, 3 movimientos 2-opt, mientras que un movimiento 5-opt se describe como una secuencia de hasta 5 movimientos 2-opt.

Para ilustrar, si describimos un movimiento 2-opt como $2\text{opt}(t_1, t_2, t_3, t_4)$, el cual intercambia las aristas (t_1, t_2) y (t_3, t_4) por (t_2, t_3) y (t_4, t_1), entonces se pueden describir movimientos de mayor orden basados en este movimiento. Por ejemplo, un movimiento 3-opt específico, $3\text{opt}(t_1, t_2, t_3, t_4, t_5, t_6)$, se puede describir como $2\text{opt}(t_1, t_2, t_4)$, luego $2\text{opt}(t_6, t_5, t_4)$ y finalmente $2\text{opt}(t_6, t_2, t_3)$.

Dado que todo se basa en el movimiento 2-opt, su implementación eficiente es fundamental. El principal costo de un movimiento 2-opt es la reversión en el sentido de recorrido del camino, lo cual en el peor de los casos tiene un costo de $O(n)$ para cada movimiento 2-opt en implementaciones *naive*. Para casos con menos de 1000 vértices, se sugiere representar un *tour* como simplemente un arreglo [7]. Sin embargo, para casos con más de 1000 vértices, como es el interés de Synopsys, se propone el uso del *two-level tree* descrito en [18]. Este enfoque se elige como la implementación a realizar, ya que tiene un costo de reversa de $O(\sqrt{n})$ por movimiento 2-opt. Como se indicará en las pautas para futuros desarrollos de Synopsys, este costo puede mejorarse aún más.

Es esencial recordar que un criterio para los movimientos es que sean secuenciales. Para lograrlo, se implementa un stack en el cual se ubica el último vértice que se incorporó a la secuencia y que será el próximo punto de inicio para agregar un siguiente vértice a la misma. A través de este stack, podemos conocer en orden cada uno de los vértices que conforman el intercambio secuencial, permitiéndonos ordenarlos según su orden de adición al intercambio y revertir elecciones si es necesario.

3.2.7. Almacenamiento de óptimos locales obtenidos

Cada aplicación del algoritmo de TSp sobre una cadena, obtiene un resultado óptimo local determinado por la primera arista extraída del grafo. Dado que se ha definido el criterio adicional de, al llegar a un óptimo local, comparar el estado actual de la solución con los óptimos locales para evitar el proceso de finalización, cada uno de los *tours* óptimos locales se almacena en una tabla hash. En esta tabla, la clave corresponde a la longitud de ese *tour*. Luego, dado un *tour*, antes de realizar el proceso de finalización, podemos determinar qué óptimos locales tienen exactamente el mismo costo mediante el uso de la función de hashing, facilitando la comparación con ese *tour* específico. Este enfoque mejora la eficiencia al evitar la repetición de cálculos costosos cuando se encuentran soluciones equivalentes.

3.2.8. Multitud de pequeñas decisiones

A continuación, se presenta una lista no exhaustiva de múltiples decisiones implementadas en el algoritmo con el objetivo de mejorar su eficiencia o resultados. Esto ilustra el nivel de detalle del algoritmo:

- Calculo de distancias euclidianas: Se implementa una cota inferior rápida para el cálculo de distancias euclidianas. Esta cota permite identificar distancias demasiado largas, las cuales no valdría la pena revisar, ahorrando así el costo computacional de calcularlas y reduciendo los costos computacionales al evitar el calculo excesivo de raíces necesario para obtener esta distancia.
- Inversión de caminos en movimientos 2-opt: Dado un movimiento 2-opt, se puede invertir el sentido de recorrido de cualquiera de los dos caminos disjuntos obtenidos al quitar las aristas. Se decide invertir el sentido del camino de menor longitud para ahorrar costos, ya que la inversión tiene un costo de $O(\sqrt{n})$ con n como la longitud del camino.
- Tabla hash para el calculo de distancias: Se utiliza una tabla hash para el cálculo de distancias en los casos de coordenadas, llenando la tabla a medida que se necesitan y se van calculando. Esto contribuye a la eficiencia al evitar cálculos redundantes a la vez que evitando calcular todas las distancias a priori.
- Selección de aristas de igual alfa-distancia: Si dos aristas tienen la misma alfa-distancia, se prefiere la que tenga el menor costo. Esta elección se realiza para favorecer la eficiencia y obtener soluciones de mejor calidad.
- Restricción en la extracción de aristas: La primera arista a extraer del *tour* no puede pertenecer a ninguno de los óptimos locales obtenidos anteriormente ya que probablemente sea parte del *tour* óptimo
- Optimización del proceso de finalización: Se intenta realizar una serie de movimientos no secuenciales 4-opt "Crossing Bridges". Estos movimientos buscan mejorar aún más la calidad del *tour* final.
- Preferencia por movimientos 5-opt: Se inicia la ejecución del algoritmo con la mayor cantidad de movimientos 5-opt para mejorar la habilidad del algoritmo para encontrar mejores *tours*. Esta elección se basa en evidencia empírica proporcionada por Helsgaun.

Entre múltiples otras decisiones, igual o más específicas, que permiten un ajuste preciso del algoritmo.

La implementación resulta en un código de más de 3000 líneas, desarrollando un programa minimalista que contiene únicamente lo esencial. Aunque este número es menor que las 4000 líneas reportadas por Helsgaun, es importante tener en cuenta que el código de Helsgaun incluye muchas más funcionalidades, como el soporte para cada uno de los casos de TSPLib, la selección de parámetros para ejecutar el código y la capacidad de definir aristas que no deben pertenecer al *tour* o forzar que algunas estén presentes, entre otras.

El objetivo de hacerlo minimalista es lograr la máxima comprensión de este algoritmo complejo para quienes permanecen en la empresa. Se implementan dos clases de inputs: una que recibe coordenadas en el espacio y su medida euclidiana para el caso simétrico, y otra que recibe matrices explícitas de costos para el caso asimétrico.

La versión para casos euclidianos tiene la importancia de servir como modelo a seguir para implementaciones futuras directamente sobre *scan flops* de dos coordenadas (una de input y otra de output). Se muestra cómo se realiza el cálculo de distancias a lo largo del proceso y cómo se podrían agregar nuevas distancias al código. Este enfoque permite una adaptación más fácil a las necesidades específicas de la empresa y demuestra flexibilidad para futuros desarrollos.

Capítulo 4

Evaluación

Como ya ha sido mencionado, las heurísticas para el Problema del Vendedor Viajero tienen como objetivo obtener soluciones lo más cercanas posible al óptimo real, al mismo tiempo que minimizar el tiempo empleado para llegar a esa solución. La evaluación de estas heurísticas consiste en medir precisamente ambas cualidades.

Ambas mediciones deben llevarse a cabo al presentar los resultados obtenidos en una amplia variedad de casos. Para cada caso, se obtiene un resultado que refleja la calidad de la solución y el tiempo empleado. Posteriormente, a través de la agregación de estos resultados en términos de calidad y tiempo, es posible realizar análisis y obtener conclusiones que caractericen el desempeño del algoritmo en cuanto a tiempo y costo.

Con el propósito de replicar los experimentos realizados en el artículo original de Helsgaun [7], se busca, en el mejor de los casos, obtener resultados lo más similares posibles. Esto permitiría afirmar que el comportamiento es similar, la implementación es correcta y, por lo tanto, las afirmaciones hechas para el algoritmo original en [7] también son aplicables a esta versión.

Por otro lado, el objetivo incluye medir resultados específicamente para el contexto de Synopsys. Como se verá a continuación, Synopsys cuenta con un conjunto de diseños de circuitos integrados, de los cuales se tienen algunas de sus particiones designadas para pruebas por ser representativas de los diseños usuales. Para estos ejemplos, también se busca medir la optimalidad y el tiempo, obteniendo así una medida concreta y objetiva del desempeño de este algoritmo. Esto establece un estándar de comparación para futuros desarrollos de Synopsys. Estos desarrollos podrán compararse con los resultados de este trabajo, facilitando la toma de decisiones para el proceso de *Reordering* basado en razones objetivas y datos concretos.

4.1. Casos de prueba de Synopsys

Synopsys cuenta con un repositorio de pruebas diseñado para poner a prueba sus algoritmos de proceso de *Reordering*. Este repositorio comprende 139 particiones derivadas de 43 diseños, cada una con un número variable de cadenas que oscila desde unas pocas unidades hasta cientos de ellas. En cuanto al número de vértices de estas cadenas, varían desde un mínimo de menos de 10 vértices hasta un máximo de algunos miles, siendo inferior a 5000

vértices la cadena de mayor tamaño.

Cada una de las cadenas que conforman la partición se encuentra inicialmente en un orden aleatorio, y el objetivo es aplicar el proceso de *Reordering*. Cabe destacar que no se conoce un óptimo exacto para este proceso, ya que nunca se han utilizado métodos exactos sobre estos casos de prueba.

Respecto a la estructura concreta de estos casos de prueba, se representa una partición en un archivo .csv. En la primera parte del archivo, cada línea corresponde a un *scan flop*. Cada una de estas líneas contiene múltiple información sobre el mismo, incluyendo el nombre de su diseño, el nombre particular de este *scan flop* y su conectividad, por ejemplo, a relojes. La información relevante para el alcance de esta memoria es extraer las coordenadas x e y del input, así como las coordenadas x e y del output.

Una vez definidos todos los *scan flops*, se definen cada una de las cadenas que componen la partición. Primero se indica el inicio de una cadena, con sus *scan flops* de inicio y final, y luego se lista el nombre de cada uno de los *scan flops* que componen la cadena.

Con el objetivo de alinear Synopsys con el estándar académico, como se ha mencionado anteriormente, especialmente para que su tipo de entrada siga el estándar de TSPLib, se ha desarrollado un parser que convierte el formato de Synopsys al formato estándar de TSPLib. Este parser determina, para cada cadena, el número de vértices del problema y su matriz explícita de costos.

El parser lee el archivo csv y calcula la distancia entre cada par de *scan flops* utilizando una función de distancia Manhattan. Luego, con estos valores, construye la matriz de costos explícita, generando así un archivo de tipo ATSP de TSPLib para cada cadena. Las particiones se corresponden con directorios, dentro de los cuales se encuentran tantos archivos ATSP como cadenas tenga la partición.

4.2. Privacidad de los datos de Synopsys

En la sección anterior, no proporcionaron valores específicos para la cantidad mínima y máxima de vértices, ni para la cantidad mínima o máxima por partición. La razón detrás de esta omisión radica en la naturaleza de Synopsys, una empresa que opera en la industria del circuito integrado. Dicha industria está actualmente bajo la atención pública debido a las crecientes tensiones de secretismo, especialmente en el contexto de los conflictos en el desarrollo de circuitos integrados entre Estados Unidos y China. Como resultado, Synopsys ejerce extrema precaución en la divulgación de información al público y solicitó que estos datos no fueran revelados.

Conforme a estas consideraciones y en consonancia con las políticas de Synopsys, los nombres asociados a los 43 diseños de circuitos integrados, que son parte integral de cada una de las 139 particiones, son tratados como información confidencial y no serán revelados.

Además, no se proporciona una descripción detallada de la implementación del parser

mencionado en la sección anterior. Este nivel de detalle se omite deliberadamente, ya que ahondar en la implementación revelaría el formato específico en el que Synopsys representa sus cadenas, y esta información se considera confidencial.

En relación a los resultados obtenidos tanto por esta memoria como por los previos, es importante destacar que esta información es altamente sensible. Incluso la revelación de resultados específicos de la longitud de cadena obtenida por el algoritmo podría llevar a la caracterización de un diseño. Esto implicaría conocer qué clientes están desarrollando qué chips en la herramienta de Synopsys y, por ende, obtener información sobre el grado de sofisticación de los algoritmos de Synopsys en comparación con la competencia.

Un argumento similar se aplica a los peligros asociados con la divulgación de los valores de las cotas inferiores determinadas para cada caso. Compartir esta información podría tener implicaciones significativas para la confidencialidad de los diseños y algoritmos empleados por Synopsys.

Por ende, se opta por no proporcionar detalles específicos sobre los resultados obtenidos, alineándose con la necesidad de mantener la confidencialidad en un entorno tan delicado y competitivo como el de la industria de circuitos integrados.

En vista de la sensibilidad de la información, los resultados se presentan de la siguiente manera:

- Cada partición será referida mediante una enumeración alternativa, utilizando la forma C_i, donde i es un número entre 1 y 139. La ordenación de las particiones se realizará de manera descendente según el número máximo de vértices en alguna de sus cadenas. Dado que el costo del algoritmo y la complejidad de resolución están vinculados al número de vértices por cadena, esta ordenación proporciona una perspectiva relevante.
- Se proporcionará el número máximo de vértices de alguna cadena dentro de la partición, normalizado a valores entre 0 y 1. Este enfoque, junto con el conocimiento de que el mínimo de aristas es menos de 10 y el máximo de aristas es menos de 5000, permitirá discutir adecuadamente la relación entre el máximo de aristas y las métricas de calidad y tiempo.
- Se omitirá la caracterización individual de cada partición, excluyendo información como el número de cadenas por partición, número de vértices promedio, mínimo y total agregado de vértices de la partición.
- No se revelarán los valores específicos de la longitud de la solución obtenida por el algoritmo implementado.
- Synopsys se abstiene de proporcionar el largo de cada partición obtenido por los algoritmos previos a la implementación de éste, no pudiéndose medir entonces el nivel de mejora obtenido por esta memoria.
- No se presentará el dato de cota inferior obtenido por este algoritmo, dada su posible cercanía al óptimo.

Lo mencionado anteriormente supone un desafío al presentar resultados de forma limitada, pero aún así es factible realizar mediciones con datos agregados o porcentuales. Estos permiten extraer conclusiones valiosas sin revelar información específica de Synopsys.

Los experimentos efectuados en esta memoria están bajo el control de Synopsys, lo que les otorga la capacidad de acceder a los valores específicos de cada partición, utilizarlos como referencia para posibles desarrollos futuros y también replicarlos.

En conclusión, en relación con el tema de la privacidad, podría haberse incluido más información en los resultados. Sin embargo, con la mayor buena fe y considerando el cuidado escrupuloso de Synopsys respecto a sus datos, se opta por sacrificar precisión y capacidad de evaluación detallada en favor de preservar la confidencialidad, aunque manteniendo la posibilidad de presentar los resultados y conclusiones fundamentales de optimalidad y costo.

4.3. Casos de prueba de TSPLib

La evaluación utilizando TSPLib posibilita la comparación entre los resultados obtenidos por el algoritmo implementado y aquellos derivados del artículo original [7], el cual es evaluado en esta librería. En caso de concordancia entre los resultados, se puede afirmar que el algoritmo sigue de manera satisfactoria la descripción proporcionada. Por lo tanto, se le pueden atribuir las características observadas por Helsgaun en su propia implementación.

La ventaja de utilizar este conjunto de datos de pruebas no solo reside en la replicación de los resultados obtenidos por Helgsaun en su artículo original, sino también en la posibilidad de extraer conclusiones que no podrían haberse obtenido exclusivamente con el conjunto de datos de prueba de Synopsys. Esto se debe tanto a las medidas de confidencialidad implementadas como a la falta de conocimiento del óptimo real en los casos de Synopsys.

En particular, el algoritmo se pone a prueba en los casos euclídeos y asimétricos de TSPLib, ampliando así el alcance de la evaluación y proporcionando un análisis más completo de su desempeño.

Caracterizando cada uno de estos conjuntos de datos de prueba, para el caso euclídeo se evalúan 61 casos. Estos casos varían en el número de vértices desde 51 hasta 4461, lo cual resulta útil ya que coincide en gran medida con las dimensiones de las cadenas de los casos de prueba de Synopsys.

En cuanto a los casos de prueba asimétricos, constan de 19 casos con cadenas que tienen un mínimo de 17 vértices y un máximo de 443. Es importante tener en cuenta que se realiza una conversión desde el caso asimétrico a un equivalente simétrico, lo cual implica una duplicación de los vértices. En efecto, el algoritmo se comporta como si se tratara de cadenas con un mínimo de 34 vértices y un máximo de 886.

4.4. Evaluación del algoritmo

A continuación, se presentan los datos de optimalidad, tiempo y cota inferior para cada uno de los casos TSP euclídeos, ATSP y casos de Synopsys. Para esto se miden las siguientes cantidades base, sobre las cuales se determinan los valores obtenidos por cada experimento:

- Tiempo de preprocesamiento por cadena
- Tiempo de Ascent por cadena
- Tiempo promedio de ejecución por cadena
- Cota inferior obtenida
- Costo mínimo obtenido por cadena dado un número ejecuciones
- Costo máximo obtenido por cadena dado un número de ejecuciones

4.4.1. Evaluación de cotas inferiores en TSPLib simétrico

Para evaluar la cota inferior del proceso de Ascent implementado, se ejecuta el algoritmo desarrollado, obteniendo la cota inferior para todos los casos de TSPLib euclídeos, las cuales se comparan porcentualmente con el valor de óptimo conocido dado por la librería. Los resultados se presentan en tablas, las cuales tiene las siguientes columnas:

- Caso: Corresponde al nombre del caso de TSPLib.
- Óptimo: Representa el óptimo conocido del caso TSPLib.
- Cota inferior: Indica la cota inferior obtenida mediante el proceso de Ascent implementado.
- Dif. %: Muestra la diferencia porcentual entre los valores de las columnas "Óptimo" y "Cota inferior" calculado como la diferencia entre los valores de "Óptimo" y "Cota inferior" sobre "Óptimo" y en formato porcentual.

Se obtienen entonces los siguientes resultados:

Tabla 4.1: Cotas inferiores obtenidas sobre casos simétricos euclídeos de TSPLib (1)

Caso	**Óptimo**	**Cota inferior**	**Dif. %**
a280	2579	2566	0.5
berlin52	7542	7542	0.0
bier127	118282	117430	0.7
ch130	6110	6075	0.7
ch150	6528	6487	0.6
d198	15780	14573	7.6
d493	35002	34820	0.5
d657	48912	48446	1.0
d1291	50801	50196	1.2
d1655	62128	61453	1.1
d2103	80450	79229	1.5
eil51	426	422	0.8
eil76	538	537	0.2
eil101	629	627	0.3
fl417	11861	11287	4.8
fl1400	20127	19532	3.0
fl1577	22249	21459	3.5
fl3795	28772	27488	4.5
fnl4461	182566	181566	0.5
gil262	2378	2354	1.0
kroA100	21282	20936	1.6
kroA150	26524	26293	0.9
kroA200	29368	29056	1.1
kroB100	22141	21832	1.4
kroB150	26130	25732	1.5
kroB200	29437	29164	0.9
kroC100	20749	20472	1.3
kroD100	21294	21142	0.7
kroE100	22068	21799	1.2
lin105	14379	14370	0.1
lin318	42029	41881	0.4
nrw1379	56638	56393	0.4

Tabla 4.2: Cotas inferiores obtenidas sobre casos simétricos euclídeos de TSPLib (2)

Caso	**Óptimo**	**Cota inferior**	**Dif. %**
p654	34643	33218	4.1
pcb442	50778	50465	0.6
pcb1173	56892	56350	1.0
pcb3038	137694	136582	0.8
pr76	108159	105051	2.9
pr107	44303	39992	9.7
pr124	59030	58061	1.6
pr136	96772	95859	0.9
pr144	58537	57876	1.1
pr152	73682	69643	5.5
pr226	80369	79448	1.1
pr264	49135	46756	4.8
pr299	48191	47378	1.7
pr439	107217	105816	1.3
pr1002	259045	256727	0.9
pr2392	378032	373488	1.2
rat99	1211	1206	0.4
rat195	2323	2292	1.3
rat575	6773	6723	0.7
rat783	8806	8772	0.4
rd100	7910	7897	0.2
rd400	15281	15156	0.8
rl1304	252948	249079	1.5
rl1323	270199	265810	1.6
rl1889	316536	311305	1.7
st70	675	671	0.6
ts225	126643	115605	8.7
tsp225	3916	3877	1.0
u159	42080	41925	0.4
u574	36905	36710	0.5
u724	41910	41649	0.6

Observamos que la diferencia porcentual promedio entre la cota inferior obtenida por el algoritmo implementado y el óptimo real del problema es del 1.66 %, lo cual está relativamente muy cerca al 1.1 % obtenido por el artículo original. Esta discrepancia podría deberse al hecho de que en esta memoria se prueban únicamente los casos euclídeos de TSPLib, mientras que en el artículo original se abarcan todos los casos de TSPLib. Es más probable que los resultados de diferencia porcentual sean más bajos cuando se tiene una mayor cantidad de casos, como se observa en las tablas. Es plausible que una mayor diversidad de casos reduzca este valor promedio.

Se observa que el error es superior al 1 % únicamente para 11 casos de los 66, lo cual indica que este promedio más alto es resultado de casos particulares en los que se obtiene un error significativo. La mayoría de los casos demuestran un bajo nivel de discrepancia porcentual, resaltándose entonces, la calidad de la cota inferior obtenida en la mayoría de los casos evaluados.

Es importante destacar que solo para un caso, la cota inferior coincide con el óptimo, específicamente en el caso "berlin52" con 52 vértices. Este hecho es excepcional, pero sin embargo demuestra la alta capacidad del algoritmo de estar cerca de óptimo real.

El hecho de haber obtenido un error porcentual promedio entre la cota inferior obtenida y el óptimo del 1.6 % es profundamente prometedor. Con apenas un poco más del 1 % de error, se puede aproximar el óptimo de una solución de manera efectiva. Esto sugiere que la cota inferior obtenida por este algoritmo puede ser utilizada como un sustituto confiable del óptimo real, especialmente en casos donde este último no ha sido calculado mediante métodos exactos. La capacidad de proporcionar una estimación cercana al óptimo real hace que la cota inferior sea una herramienta valiosa en la resolución de problemas de optimización.

El hecho mencionado anteriormente se aprovecha entonces para medir la calidad de la solución en los casos de prueba de Synopsys. Más adelante, al comparar la solución encontrada en los casos de Synopsys, podemos compararlo con la cota inferior obtenida, y entonces afirmar que el valor porcentual de error entre la solución encontrada por el algoritmo y la cota inferior es muy cercana a la que se obtendría al comparar la solución del algoritmo con el óptimo real.

Intentando establecer una relación entre el número de vértices y la diferencia porcentual obtenida, se observa fácilmente que no parece haber correlación entre ambas cantidades. Por ejemplo, se obtiene un error del 0.5 % para el caso de mayor tamaño probado, que tiene 4461 vértices, mientras que el máximo error se obtiene para una instancia de apenas 107 vértices. Este análisis sugiere que la complejidad del problema no está directamente relacionada con la magnitud de la diferencia porcentual, y otros factores pueden influir en la calidad de la cota inferior obtenida.

4.4.2. Evaluación de optimalidad y costo en TSPLib simétrico

Para evaluar el grado de calidad de la solución, se obtiene la solución del algoritmo para todos los casos de TSPLib euclídeos, los cuales se comparan porcentualmente con el valor óptimo conocido. Con el fin de poder definir la probabilidad con la cual, a partir de un *tour* inicial aleatorio, se encuentra un *tour* óptimo, se realizan 100 ejecuciones del algoritmo, obteniendo la frecuencia relativa de ejecuciones en las que se alcanza el óptimo. Adicionalmente, con el objetivo de caracterizar el costo en tiempo del algoritmo, se mide el tiempo promedio por ejecución para cada cadena.

Los resultados se presentan en la tabla, la cual tiene las siguientes columnas:

- Caso: Corresponde al nombre del caso de TSPLib.
- Aciertos: Frecuencia de ejecuciones, de un total de 100, en las que se obtiene el óptimo conocido.
- T. Prom. Ejec: Para cada una de las 100 ejecuciones, se obtiene el tiempo en segundos que demora la ejecución del algoritmo (sin contar el proceso de Ascent), luego se calcula el promedio de estas cantidades.
- Óptimo: Corresponde al óptimo conocido del caso TSPLib.
- Dif. % Prom.: Diferencia porcentual promedio del resultado obtenido en cada una de las ejecuciones con respecto al óptimo real, calculada como la diferencia entre "Óptimo" y "Largo de solución obtenida" sobre "Óptimo" y en formato porcentual.
- Dif. % Max.: Máxima diferencia obtenida entre la solución encontrada por alguna ejecución del algoritmo al compararla con el óptimo. Esta medición nos permite conocer cómo se comporta el algoritmo en el peor caso si solo lo ejecutamos una vez.

Se obtienen entonces los siguientes resultados

Tabla 4.3: Resultados de optimalidad sobre casos simétricos euclídeos de TSPLib (1)

Caso	**Aciertos**	**T. Prom. Ejec.**	**Óptimo**	**Dif. % Prom.**	**Dif. % Max**
a280	100/100	0.01	2579	0.000	0.000
berlin52	1/1	0.00	7542	0.000	0.000
bier127	100/100	0.00	118282	0.000	0.000
ch130	100/100	0.00	6110	0.000	0.000
ch150	62/100	0.03	6528	0.029	0.077
d198	100/100	0.10	15780	0.000	0.000
d493	97/100	0.28	35002	0.001	0.094
d657	0/100	1.34	48912	0.002	0.002
d1291	72/100	14.66	50801	0.042	0.167
d1655	99/100	3.62	62128	0.000	0.002
d2103	0/100	91.91	80450	0.037	0.089
eil51	100/100	0.00	426	0.000	0.000
eil76	100/100	0.00	538	0.000	0.000
eil101	100/100	0.00	629	0.000	0.000
fl417	92/100	0.91	11861	0.035	0.455
fl1400	0/100	62.24	20127	0.194	0.199
fl1577	44/100	97.20	22249	0.032	0.067
fl3795	29/100	354.81	28772	0.052	0.379
fnl4461	46/100	82.82	182566	0.002	0.005
gil262	100/100	0.05	2378	0.000	0.000
kroA100	100/100	0.00	21282	0.000	0.000
kroA150	100/100	0.01	26524	0.000	0.000
kroA200	100/100	0.02	29368	0.000	0.000
kroB100	100/100	0.01	22141	0.000	0.000
kroB150	50/100	0.07	26130	0.004	0.008
kroB200	100/100	0.01	29437	0.000	0.000
kroC100	100/100	0.00	20749	0.000	0.000
kroD100	100/100	0.00	21294	0.000	0.000
kroE100	98/100	0.02	22068	0.003	0.172
lin105	100/100	0.00	14379	0.000	0.000
lin318	63/100	0.14	42029	0.096	0.271
nrw1379	52/100	3.78	56638	0.004	0.009

Tabla 4.4: Resultados de optimalidad sobre casos simétricos euclídeos de TSPLib (1)

Caso	**Aciertos**	**T. Prom. Ejec.**	**Óptimo**	**Dif. % Prom.**	**Dif. % Max**
p654	100/100	0.94	34643	0.000	0.000
pcb442	93/100	0.33	50778	0.001	0.014
pcb1173	81/100	1.30	56892	0.002	0.009
pcb3038	70/100	37.25	137694	0.007	0.048
pr76	100/100	0.01	108159	0.000	0.000
pr107	100/100	0.00	44303	0.000	0.000
pr124	100/100	0.01	59030	0.000	0.000
pr136	100/100	0.02	96772	0.000	0.000
pr144	100/100	0.01	58537	0.000	0.000
pr152	100/100	0.03	73682	0.000	0.000
pr226	100/100	0.01	80369	0.000	0.000
pr264	100/100	0.04	49135	0.000	0.000
pr299	100/100	0.08	48191	0.000	0.000
pr439	99/100	0.12	107217	0.000	0.041
pr1002	99/100	0.34	259045	0.000	0.001
pr2392	100/100	3.27	378032	0.000	0.000
rat99	100/100	0.0	1211	0.000	0.000
rat195	100/100	0.03	2323	0.000	0.000
rat575	71/100	0.66	6773	0.004	0.015
rat783	100/100	0.03	8806	0.000	0.000
rd100	100/100	0.00	7910	0.000	0.000
rd400	99/100	0.10	15281	0.000	0.007
rl1304	53/100	2.31	252948	0.075	0.329
rl1323	13/100	3.89	270199	0.014	0.048
rl1889	48/100	9.95	316536	0.024	0.084
st70	100/100	0.00	675	0.000	0.000
ts225	100/100	0.01	126643	0.000	0.000
tsp225	100/100	0.01	126643	0.000	0.000
u159	100/100	0.00	42080	0.000	0.000
u574	97/100	0.24	36905	0.002	0.081
u724	100/100	0.29	41910	0.000	0.000

Se obtiene que el promedio del error máximo, dado como el promedio de la columna Dif. % Max., es 0.04176, mientras que el error promedio entre todas las particiones, dado como el promedio de la columna Dif. % Prom., es 0.01034 %, lo cual es prometedor, por lo bajo que es.

El tiempo para la gran mayoría de los casos es de menos de un segundo, incluso menos de una décima de segundo en muchos casos, y llegando incluso a valores mejores a una centésima de segundo, siendo aquellos en los que se indica tiempo 0.00. Este rendimiento en tiempo

sugiere una eficiencia notable del algoritmo implementado en la resolución de los casos de prueba evaluados.

Se observa una tremenda variabilidad en el tiempo de ejecución de los distintos casos de prueba, los cuales, al igual que con la cota inferior, no parecen ser consecuencia del número máximo de vértices por caso. Por ejemplo, el caso “fnl4461” con 4461 vértices demora 82.82 segundos, mientras que un caso con menos vértices, a ser “fl3795”, demora 354.81 segundos, más de cuatro veces el tiempo obtenido.

Sin embargo, es una regla general, al menos sobre TSPLib, que los casos con más de 1000 vértices toman más de un segundo. La variabilidad en los tiempos de ejecución podría deberse a diversas características y complejidades intrínsecas de cada instancia del problema, y no se correlaciona directamente con el número de vértices. Este comportamiento coincide con el observado en el articulo original, afirmándose que “es difícil predecir los tiempos de ejecución necesarios para resolver un problema dado”

Se tiene el caso particular de "fl3795"que toma casi seis minutos. Realmente, considerando la complejidad del problema y que sea un solo caso aislado, seis minutos resulta razonable. Además, teniendo en cuenta que se obtiene un error de apenas 0.379 %, siendo indiscutiblemente el peor caso obtenido, la inversión de tiempo parece justificada por la calidad de la solución obtenida.

En particular, interesa el caso de la diferencia máxima, ya que esto representa el peor escenario al ejecutar una sola ejecución. Obtener un error promedio de 0.04176 % en una sola ejecución es particularmente útil y rápido, especialmente considerando que, como ya fue mencionado, la mayoría de las ejecuciones toma menos de un segundo.

Vale la pena mencionar el caso de “berlin52”, ya que al igual que en el artículo original, se verifica que el proceso de Ascent obtiene un 1-tree con el mismo largo y, a la vez, correspondiendo al óptimo. Por ello, se indica como aciertos 1/1, destacando la capacidad del algoritmo para encontrar la solución óptima en este caso específico.

Vale la pena mencionar el caso "d2103". Este caso no está presente en el paper original de Helsgaun, por lo que es probable que haya sido agregado a TSPLib después del año 2000. Para este caso, se obtiene que no se puede determinar el óptimo. Sin embargo, aunque esta información nos brinda claridad, no significa que el algoritmo haya sido implementado incorrectamente, ya que este resultado podría haber sido similar para el algoritmo original.

Se presentan las siguientes diferencias específicas con los experimentos originales de Helsgaun:

- En el caso “d493” se obtienen 97 aciertos, cifra inferior a los 100 aciertos obtenidos por el algoritmo original.
- En el caso "d657"no se obtiene ningún acierto, a diferencia del algoritmo original que logra 100 aciertos. Sin embargo, vale la pena mencionar que se obtiene un error máximo de 0.002 % respecto al óptimo y un promedio de igual valor. Esto sugiere que probable-

mente el algoritmo siempre llegó a un mismo valor en todas sus ejecuciones, un valor que se sitúa un 0.002 % por encima del óptimo real.

- En el caso "d1655"se obtienen 99 aciertos, mientras que el algoritmo original alcanza el óptimo en todas sus ejecuciones.
- Para el caso "fl417"se obtienen 92 aciertos, en contraste con los 88 aciertos del artículo original.
- En el caso "fl1400"no se obtienen aciertos, a diferencia del artículo original donde se logra un acierto tras 10 ejecuciones.
- En el caso "kroE100"se obtienen 98 aciertos, en contraste con los 99 aciertos del artículo original.
- Para el caso "kroB150"se obtienen 50 aciertos, en contraste con los 55 aciertos del artículo original.

Y así, múltiples otras diferencias, pero como se puede evidenciar, son insignificantes, apenas un poco diferentes, pudiendo considerarse entonces que este algoritmo conserva la calidad del artículo original con resultados prácticamente iguales.

Se observa que los resultados en los casos euclídeos de TSPLib son casi exactamente los obtenidos en el paper original. Se puede afirmar entonces, en base a la evidencia empírica, que este algoritmo desarrollado tiene la misma calidad que el desarrollado por Helsgaun en su paper [7]. Esto no es de extrañar, ya que su guía detallada a través de las 71 páginas de su artículo, incluso detallando aspectos de desarrollo, resulta fundamental para esta implementación.

Con esta similitud, entonces se puede afirmar con propiedad que esta implementación ha de cumplir las características descritas en el artículo original, con apenas algunas diferencias.

Como recapitulación de lo obtenido en esta evaluación, se obtiene una calidad de solución en el peor caso de un promedio del 0.04176 %, lo cual es altamente prometedor para Synopsys y es prácticamente óptimo. Además, en la gran mayoría de los casos, para instancias de menos de 1000 vértices, el resultado se obtiene en mucho menos de un segundo.

4.4.3. Evaluación de optimalidad en TSPLib asimétrico

Se realiza el mismo procedimiento antes realizado con la optimalidad en casos de prueba simétricos euclídeos, pero ahora con los casos asimétricos de TSPLib. El fin fundamental de esta evaluación es simplemente evidenciar si el comportamiento sobre archivos de entrada matriciales explícitos en formato ATSP es correctamente ejecutado y los resultados son correctos.

Las conclusiones respecto a tiempo, optimalidad y cotas inferiores de este programa ya fueron obtenidas previamente con los casos euclídeos simétricos, caso que es la base del programa. Aquí únicamente se prueba una conversión correcta y la capacidad de trabajar sobre una entrada diferente en formato estándar de casos simétricos de TSPLib.

Las columnas se describen de la misma manera que en el caso euclídeo simétrico.

Los resultados son los siguientes

Tabla 4.5: Resultados sobre casos asimétricos de TSPLib

Caso	**Aciertos**	**T. Prom. Ejec.**	**Óptimo**	**Dif. % Prom.**	**Dif. % Max**
br17	100/100	0.00	39	0.000	0.000
ft53	100/100	0.00	6905	0.000	0.000
ft70	100/100	0.00	38673	0.000	0.000
ftv33	100/100	0.00	1286	0.000	0.000
ftv35	72/100	0.00	1475	0.038	0.136
ftv38	91/100	0.00	1530	0.012	0.131
ftv44	100/100	0.00	1613	0.000	0.000
ftv47	100/100	0.00	1776	0.000	0.000
ftv55	100/100	0.00	1608	0.000	0.000
ftv64	100/100	0.00	1839	0.000	0.000
ftv70	100/100	0.00	1950	0.000	0.000
ftv170	100/100	0.02	2755	0.000	0.000
kro124p	100/100	0.00	36230	0.000	0.000
p43	0/100	0.02	5620	0.018	0.018
rbg323	30/100	3.53	1326	0.106	0.226
rbg358	97/100	1.79	1163	0.007	0.516
rbg403	100/100	0.07	2465	0.000	0.000
rbg443	100/100	0.06	2720	0.000	0.000
ry48p	100/100	0.00	14422	0.000	0.000

Vale la pena notar que para todos los ejemplos de TSPLib se obtiene el óptimo en un altísimo porcentaje de ejecuciones. Es relevante centrar la atención en el caso de "p43". En este caso, el óptimo reportado por la librería TSPLib es de 5620, mientras que el algoritmo desarrollado genera un costo de 5621, apenas una unidad de diferencia. Por lo tanto, aunque no se haya obtenido nunca el costo óptimo, en realidad el 100 % de las ejecuciones obtiene 5621.

En base a estos resultados y al ser prácticamente idénticos a los reportados por Helsgaun, se concluye que la conversión desde casos asimétricos a simétricos fue implementada correctamente y que el programa trabaja adecuadamente sobre casos de prueba en formato estándar ATSP de TSPLib.

4.4.4. Evaluación de optimalidad en diseños de Synopsys

A continuación, se presentan los resultados obtenidos para la optimalidad en los diseños y casos de prueba de Synopsys. Dado que no se tiene calculado de forma exacta el óptimo de estos casos y considerando que el error promedio en la cota inferior es del 1.6 % para TSPLib, se utilizan estos valores como referencia de optimalidad, comparando las soluciones obtenidas con el valor de la cota inferior.

Anteriormente, al evaluar la optimalidad del algoritmo en comparación con el caso euclideo para 100 ejecuciones, se observó que la diferencia entre la diferencia porcentual máxima y promedio entre 100 ejecuciones era muy cercana. Esto sugiere que las ejecuciones para cadenas de menos de 5000 vértices son muy similares entre sí y que el beneficio en términos de ahorro de tiempo por una única ejecución es mucho mayor que la mejora de optimalidad obtenida por más ejecuciones.

Para evaluar lo anterior de manera empírica, se realizan 1 ejecución y 10 ejecuciones, midiendo la calidad de la solución obtenida en cada caso.

En particular, se miden y definen las siguientes cantidades:

- Máx. Dimensión: Corresponde a la normalización entre 0 y 1 de la cantidad de vértices máxima de cadena para cada partición.
- Dif. % lb-lp 1 Ejec.: Corresponde a la diferencia porcentual entre la cota inferior de cada partición y su longitud mínima obtenida tras una sola ejecución.
- Dif. % lb-lp 10 Ejec.: Análogo a Dif. % lb-lp 1 Ejec., pero con 10 ejecuciones del algoritmo en vez de una.

Tabla 4.6: Resultados de optimalidad sobre casos de prueba de Synopsys (1)

Caso	**Máx. Dimensión**	**Dif. % lb-lp 1 Ejec.**	**Dif. % lb-lp 10 Ejec.**
C1	1	6.59	x
C2	0.547836685	4.48	x
C3	0.486593541	2.88	x
C4	0.35984156	24.69	x
C5	0.300731261	3.73	x
C6	0.300121877	1.79	1.76
C7	0.28031688	1.27	1.23
C8	0.265082267	2.60	2.53
C9	0.256855576	1.64	1.50
C10	0.256246191	1.67	1.66
C11	0.255941499	2.43	2.39
C12	0.255941499	1.31	1.30
C13	0.255636807	2.54	2.53
C14	0.253503961	11.04	11.04
C15	0.246191347	1.03	1.03
C16	0.221511274	5.00	4.98
C17	0.198659354	1.41	1.40
C18	0.195612431	2.93	2.92
C19	0.18738574	7.50	7.49
C20	0.187081048	17.16	17.16
C21	0.153564899	10.06	10.00
C22	0.134978672	3.69	3.65
C23	0.134978672	2.99	2.99
C24	0.132541133	2.44	2.38
C25	0.123400366	2.26	2.22
C26	0.118829982	4.06	4.03
C27	0.118829982	2.21	2.21
C28	0.118829982	1.82	1.82
C29	0.118829982	3.40	3.38
C30	0.118829982	3.62	3.60
C31	0.118829982	1.09	1.08
C32	0.118829982	1.41	1.41
C33	0.118829982	2.43	2.38
C34	0.118525289	1.14	1.14
C35	0.118525289	2.30	2.27

Tabla 4.7: Resultados de optimalidad sobre casos de prueba de Synopsys (2)

Caso	**Máx. Dimensión**	**Dif. % lb-lp 1 Ejec.**	**Dif. % lb-lp 10 Ejec.**
C36	0.118525289	3.08	3.06
C37	0.118525289	2.28	2.27
C38	0.118525289	1.49	1.49
C39	0.118525289	2.28	2.26
C40	0.118525289	2.28	2.26
C41	0.118525289	2.39	2.39
C42	0.118220597	1.46	1.42
C43	0.118220597	1.97	1.95
C44	0.110907983	1.78	1.73
C45	0.110907983	1.78	1.73
C46	0.109993906	3.44	3.43
C47	0.109384522	0.89	0.89
C48	0.109384522	0.89	0.89
C49	0.109384522	2.15	2.10
C50	0.109384522	2.10	2.06
C51	0.109384522	2.15	2.10
C52	0.109384522	2.10	2.06
C53	0.10786106	1.87	1.86
C54	0.085313833	2.41	2.39
C55	0.077087142	3.06	3.03
C56	0.077087142	3.06	3.03
C57	0.077087142	5.47	5.42
C58	0.077087142	5.47	5.42
C59	0.07678245	1.24	1.23
C60	0.076477757	2.52	2.50
C61	0.074649604	1.82	1.78
C62	0.074344912	2.77	2.73
C63	0.074344912	2.77	2.73
C64	0.074040219	2.37	2.35
C65	0.074040219	1.49	1.48
C66	0.069165143	0.18	0.18
C67	0.066727605	2.52	2.49
C68	0.066727605	2.52	2.49
C69	0.065813528	2.40	2.39
C70	0.059110299	2.80	2.78

Tabla 4.8: Resultados de optimalidad sobre casos de prueba de Synopsys (3)

Caso	**Máx. Dimensión**	**Dif. % lb-lp 1 Ejec.**	**Dif. % lb-lp 10 Ejec.**
C71	0.059110299	2.80	2.78
C72	0.05393053	3.27	3.25
C73	0.048750762	2.24	2.22
C74	0.046008531	4.78	4.75
C75	0.046008531	7.68	7.68
C76	0.046008531	7.68	7.68
C77	0.043570993	28.42	28.42
C78	0.036563071	3.13	3.12
C79	0.036563071	2.78	2.75
C80	0.036258379	3.66	3.65
C81	0.03229738	0.21	0.21
C82	0.031687995	13.84	13.82
C83	0.031078611	3.06	3.05
C84	0.030773918	13.97	13.96
C85	0.02559415	1.10	1.10
C86	0.02559415	1.10	1.10
C87	0.024984765	1.01	1.01
C88	0.024984765	1.94	1.92
C89	0.024984765	1.94	1.92
C90	0.024680073	1.68	1.68
C91	0.024375381	1.34	1.34
C92	0.024375381	1.34	1.34
C93	0.024070689	5.70	5.70
C94	0.023156612	1.40	1.40
C95	0.02285192	0.54	0.54
C96	0.02285192	0.54	0.54
C97	0.020414381	41.85	41.85
C98	0.015234613	2.44	2.21
C99	0.015234613	2.44	2.21
C100	0.014625229	7.87	7.87
C101	0.013101767	0.00	0.00
C102	0.012492383	11.35	11.35
C103	0.011882998	2.43	2.43
C104	0.011578306	0.03	0.03
C105	0.011578306	0.38	0.38

Tabla 4.9: Resultados de optmalidad sobre casos de prueba de Synopsys (4)

Caso	**Máx. Dimensión**	**Dif. % lb-lp 1 Ejec.**	**Dif. % lb-lp 10 Ejec.**
C106	0.010664229	4.88	4.88
C107	0.010664229	0.26	0.26
C108	0.010359537	0.05	0.05
C109	0.009750152	0.39	0.39
C110	0.00944546	0.35	0.35
C111	0.00944546	0.07	0.07
C112	0.007007922	0.14	0.14
C113	0.00670323	0.00	0.00
C114	0.00670323	1.12	1.12
C115	0.005789153	39.84	39.84
C116	0.005789153	0.16	0.16
C117	0.004265692	0.00	0.00
C118	0.004265692	0.41	0.41
C119	0.004265692	0.41	0.41
C120	0.00274223	0.00	0.00
C121	0.00274223	0.00	0.00
C122	0.002437538	0.00	0.00
C123	0.002437538	0.00	0.00
C124	0.002437538	0.05	0.05
C125	0.002437538	0.05	0.05
C126	0.002437538	0.05	0.05
C127	0.001828154	0.01	0.01
C128	0.001828154	0.01	0.01
C129	0.001828154	0.01	0.01
C130	0.001828154	0.00	0.00
C131	0.001828154	0.00	0.00
C132	0.001523461	0.02	0.02
C133	0.001523461	0.02	0.02
C134	0.001523461	0.02	0.02
C135	0.001218769	0.00	0.00
C136	0.001218769	0.00	0.00
C137	0.001218769	0.00	0.00
C138	0	0	0.00
C139	0	0	0.00

Aunque los datos específicos de Synopsys sean confidenciales y se reserven los valores obtenidos previos a esta memoria, se indica que se obtienen resultados mejores en todas las instancias.

La obtención de resultados superiores en todos los casos de prueba de Synopsys era un resultado esperado, dado que el algoritmo implementado corresponde al implementado por Helsgaun. Como se evidenció en la revisión bibliográfica, el grado de errores de Lin-Kernighan ya era menor que Nearest Neighbors y 2-opt, actualmente implementados en Synopsys. Por lo tanto, al ser la versión de Helsgaun mejor en optimalidad en comparación con Lin-Kernighan, por transitividad debería ser mejor que el algoritmo anterior de Synopsys. En este sentido, no es una sorpresa.

Midiendo la optimalidad de la solución como la diferencia porcentual entre la cota inferior de la partición, es decir, la sumatoria de las cotas inferiores de cada una de las cadenas que la componen, y comparándola con el largo total de la partición (que corresponde a la sumatoria de los largos de las soluciones para todas las cadenas que componen la partición), se obtiene que para una ejecución se tiene un error porcentual de 3.15 %, mientras que para 10 ejecuciones se obtiene un error de 3.13 % en el mejor caso. Esto implica una mejora de solo 0.02 %, lo cual claramente no justifica multiplicar el tiempo de ejecución por 10, confirmando las sospechas iniciales sobre el probable bajo beneficio de aumentar el número de ejecuciones.

La variación en el grado de calidad de la solución entre realizar una ejecución o diez es apenas del orden de 0.01 % en comparación con la mayor cota inferior dada por el proceso de Ascent, con una máxima diferencia de 0.07 %.

Este fenómeno puede explicarse por el hecho de que la mayoría de las cadenas son cortas y que cuanto más corta es una cadena, es más probable que se obtenga el óptimo real en una ejecución cualquiera. Por lo tanto, en la mayoría de las cadenas, se obtiene el óptimo únicamente con una ejecución, y este resultado no cambia al ejecutar 10 ejecuciones. Solo se corrigen algunas de las cadenas, de las cuales se obtiene el óptimo después de 10 ejecuciones.

En base a lo anterior, se determina que el algoritmo debería utilizarse con únicamente una ejecución ya que el aumentar el número de ejecuciones genera un beneficio mínimo y un tiempo de ejecución de aumento lineal al número de ejecuciones decidido

Con respecto al grado de optimalidad en relación con el tamaño de la instancia, nuevamente, al igual que en las cantidades anteriores de cota inferior y optimalidad para casos de prueba de TSPLib, no se observa una relación directa y clara entre ambas cantidades. Por ejemplo, para el caso con el mayor número de vértices, se obtiene un error porcentual de 6.59 % con respecto a la cota inferior, mientras que para el cuarto caso con el máximo número de vértices por cadena, el error asciende a 24.69 %, y luego baja nuevamente a 3 % para el siguiente caso. Esto evidencia un patrón absolutamente errático, lo que sugiere que la complejidad del problema no está determinada únicamente por el número de vértices, sino también por la distribución particular de esos vértices en el espacio y la naturaleza específica de su óptimo.

Realmente, el valor de la máxima dimensión normalizado queda como referencia para

observar los resultados e intentar extraer conclusiones respecto a cómo se distribuyen los óptimos. Después de realizar esta experimentación, queda aún más evidente que el largo de la cadena no es una cantidad que permita conocer directamente el comportamiento del problema en términos de optimalidad.

En base a lo anterior, se llega a la conclusión de que el número de vértices no es un indicador absoluto, y más que intentar establecer relaciones absolutas, lo mejor es simplemente comparar versiones anteriores y resultados anteriores con resultados nuevos. Si un algoritmo obtiene mejores resultados que otro sobre el mismo conjunto de datos de prueba, se puede decir que este algoritmo es mejor. Por esta razón, TSPLib se vuelve tan importante al ser el estándar más extendido.

4.5. Evaluación de tiempos en diseños de Synopsys

Aunque sea evidente a estas alturas que es suficiente y, de hecho, deseable solo utilizar una ejecución del algoritmo, se presentan de todas formas los resultados de tiempo obtenidos para proporcionar una idea de los tiempos esperados durante la ejecución de este programa.

Es importante mencionar que el tiempo en estos casos no solo depende del número de vértices de la cadena más larga, sino también del número de cadenas por partición. Hay particiones con apenas unas cuantas cadenas, mientras que otras tienen cientas.

Las cantidades medidas corresponden a las siguientes

- Caso: Corresponde al nombre del caso de Synopsys según la enumeración ya explicada para guardar confidencialidad.
- T. Prep. C.: Al tratarse de casos asimétricos, para cada cadena es necesaria la conversión desde su matriz asimétrica explícita a su matriz simétrica equivalente.
- T. Ascent C.: Tiempo promedio que toma el Ascent por cada cadena.
- T. Ejec. C.: Tiempo promedio por ejecución por cadena.
- T. Total C.: Tiempo total promedio por cadena.
- T. Total P.: Tiempo total que toma resolver la partición completamente, es decir, resolver cada una de las cadenas que la componen.

Vale mencionar que cada una de estas cantidades temporales se mide en segundos.

A continuación se presentan los resultados obtenidos:

Tabla 4.10: Resultados de tiempo sobre los casos de prueba de Synopsys (1)

Caso	**T. Prep. C**	**T. Ascent C.**	**T. Ejec. C.**	**T. Total C.**	**T. Total P.**
C1	40.81	38.70	356.60	436.11	800.54
C2	14.33	13.73	84.58	112.64	299.22
C3	1.99	1.90	11.84	15.73	1021.05
C4	4.40	4.15	47.60	56.15	52.17
C5	3.03	2.88	19.19	25.10	1123.17
C6	1.73	1.65	8.63	12.01	336.10
C7	2.45	2.31	13.78	18.54	131.48
C8	1.89	1.81	10.76	14.46	256.06
C9	2.07	1.96	10.49	14.52	127.42
C10	2.25	2.15	10.84	15.24	331.73
C11	1.98	1.89	9.40	13.27	253.80
C12	2.19	2.09	10.56	14.84	323.27
C13	1.17	1.10	7.14	9.41	16.84
C14	1.49	1.42	6.72	9.63	33.26
C15	1.80	1.72	9.64	13.16	23.18
C16	1.61	1.53	8.93	12.07	32.09
C17	1.23	1.17	5.53	7.93	220.11
C18	1.19	1.13	5.14	7.46	206.05
C19	1.71	1.65	6.02	9.38	7.89
C20	1.08	1.03	8.73	10.84	9.93
C21	0.83	0.79	4.00	5.62	4.91
C22	0.68	0.66	2.72	4.06	201.03
C23	0.44	0.41	1.63	2.48	6.36
C24	0.67	0.65	2.86	4.18	129.54
C25	0.26	0.25	0.88	1.39	112.95
C26	0.55	0.53	1.91	2.99	241.98
C27	0.54	0.52	1.76	2.82	33.02
C28	0.46	0.44	1.51	2.41	12.08
C29	0.38	0.36	1.82	2.56	22.52
C30	0.43	0.41	2.22	3.06	13.51
C31	0.45	0.43	1.68	2.56	19.63
C32	0.48	0.46	1.79	2.73	34.97
C33	0.43	0.41	1.49	2.33	13.91
C34	0.54	0.52	1.80	2.86	23.98
C35	0.48	0.46	1.83	2.77	21.25

Tabla 4.11: Resultados de tiempo sobre los casos de prueba de Synopsys (2)

Caso	**T. Prep. C**	**T. Ascent C**	**T. Ejec. C.**	**T. Total C.**	**T. Total P.**
C36	0.47	0.45	2.01	2.93	12.67
C37	0.51	0.49	1.66	2.66	22.24
C38	0.41	0.39	1.52	2.32	13.86
C39	0.49	0.47	1.81	2.77	96.42
C40	0.49	0.47	1.83	2.79	97.46
C41	0.53	0.51	1.81	2.85	98.09
C42	0.50	0.47	2.01	2.98	105.39
C43	0.49	0.47	1.72	2.68	92.86
C44	0.40	0.38	1.08	1.86	22.90
C45	0.41	0.39	1.09	1.89	23.17
C46	0.38	0.36	1.36	2.10	16.11
C47	0.23	0.22	0.59	1.04	4.32
C48	0.25	0.23	0.62	1.10	4.47
C49	0.42	0.40	1.64	2.46	218.66
C50	0.44	0.42	1.64	2.50	76.70
C51	0.41	0.39	1.64	2.44	218.28
C52	0.43	0.41	1.62	2.46	75.77
C53	0.18	0.17	0.46	0.81	65.18
C54	0.15	0.14	0.46	0.75	20.61
C55	0.24	0.23	0.77	1.24	771.52
C56	0.24	0.23	0.77	1.24	773.11
C57	0.24	0.23	0.78	1.25	385.74
C58	0.25	0.24	0.78	1.27	386.88
C59	0.20	0.19	0.51	0.90	48.16
C60	0.25	0.23	0.71	1.19	192.30
C61	0.24	0.23	0.72	1.19	7.00
C62	0.05	0.04	0.08	0.17	47.60
C63	0.04	0.04	0.07	0.15	41.48
C64	0.23	0.22	0.71	1.16	186.15
C65	0.19	0.18	0.43	0.80	4.58
C66	0.16	0.15	0.14	0.45	0.33
C67	0.09	0.08	0.17	0.34	200.05
C68	0.09	0.08	0.17	0.34	199.74
C69	0.05	0.05	0.09	0.19	15.77
C70	0.08	0.07	0.14	0.29	121.49

Tabla 4.12: Resultados de tiempo sobre los casos de prueba de Synopsys (3)

Caso	**T. Prep. C**	**T. Ascent C**	**T. Ejec. C.**	**T. Total C.**	**T. Total P.**
C71	0.08	0.08	0.15	0.31	127.12
C72	0.08	0.08	0.14	0.30	164.45
C73	0.06	0.05	0.09	0.20	5.30
C74	0.07	0.07	0.14	0.28	56.12
C75	0.04	0.04	0.13	0.21	1.70
C76	0.04	0.04	0.14	0.22	1.86
C77	0.08	0.08	0.18	0.34	0.28
C78	0.06	0.05	0.09	0.20	62.07
C79	0.06	0.06	0.11	0.23	3.33
C80	0.06	0.06	0.09	0.21	1.58
C81	0.04	0.04	0.04	0.12	0.45
C82	0.06	0.05	0.09	0.20	4.83
C83	0.04	0.04	0.06	0.14	90.64
C84	0.05	0.05	0.08	0.18	3.27
C85	0.02	0.01	0.01	0.04	0.29
C86	0.02	0.02	0.01	0.05	0.30
C87	0.05	0.04	0.04	0.13	0.10
C88	0.03	0.03	0.03	0.09	27.24
C89	0.03	0.03	0.03	0.09	27.17
C90	0.02	0.02	0.03	0.07	0.29
C91	0.01	0.01	0.02	0.04	0.23
C92	0.01	0.01	0.02	0.04	0.26
C93	0.01	0.01	0.02	0.04	0.15
C94	0.03	0.02	0.02	0.07	0.06
C95	0.01	0.01	0.01	0.03	0.20
C96	0.02	0.01	0.01	0.04	0.21
C97	0.02	0.02	0.07	0.11	0.10
C98	0.01	0.01	0.00	0.02	0.08
C99	0.01	0.01	0.00	0.02	0.09
C100	0.01	0.01	0.01	0.03	0.06
C101	0.01	0.01	0.00	0.02	0.03
C102	0.01	0.01	0.01	0.03	0.04
C103	0.01	0.01	0.01	0.03	0.05
C104	0.02	0.02	0.00	0.04	0.04
C105	0.01	0.01	0.01	0.03	0.04

Tabla 4.13: Resultados de tiempo sobre los casos de prueba de Synopsys (4)

Caso	T. Prep. C	T. Ascent C	T. Ejec. C.	T. Total C.	T. Total P.
C106	0.01	0.01	0.01	0.03	0.06
C107	0.01	0.01	0.00	0.02	0.04
C108	0.01	0.01	0.00	0.02	0.04
C109	0.01	0.00	0.00	0.01	0.05
C110	0.01	0.00	0.00	0.01	0.03
C111	0.01	0.01	0.00	0.02	0.03
C112	0.00	0.00	0.00	0.00	2.98
C113	0.00	0.00	0.00	0.00	0.03
C114	0.00	0.00	0.00	0.00	0.02
C115	0.00	0.00	0.00	0.00	0.02
C116	0.00	0.00	0.00	0.00	1.32
C117	0.00	0.00	0.00	0.00	0.02
C118	0.00	0.00	0.00	0.00	0.09
C119	0.00	0.00	0.00	0.00	0.04
C120	0.00	0.00	0.00	0.00	0.23
C121	0.00	0.00	0.00	0.00	0.16
C122	0.00	0.00	0.00	0.00	0.27
C123	0.00	0.00	0.00	0.00	0.21
C124	0.00	0.00	0.00	0.00	0.24
C125	0.00	0.00	0.00	0.00	0.14
C126	0.00	0.00	0.00	0.00	0.14
C127	0.00	0.00	0.00	0.00	0.16
C128	0.00	0.00	0.00	0.00	0.12
C129	0.00	0.00	0.00	0.00	0.12
C130	0.00	0.00	0.00	0.00	0.12
C131	0.00	0.00	0.00	0.00	0.08
C132	0.00	0.00	0.00	0.00	0.41
C133	0.00	0.00	0.00	0.00	0.27
C134	0.00	0.00	0.00	0.00	0.27
C135	0.00	0.00	0.00	0.00	0.24
C136	0.00	0.00	0.00	0.00	0.17
C137	0.00	0.00	0.00	0.00	0.02
C138	0.00	0.00	0.00	0.00	0.02
C139	0.00	0.00	0.00	0.00	0.01

Parte fundamental y crítica de este trabajo es la relación de *trade-off* entre el tiempo y la calidad de la solución, buscando minimizar el tiempo de ejecución.

Dado que el costo del algoritmo es de $O(n^{2.2})$ en el caso promedio, se podría entonces inferir que el tiempo por ejecución es casi directamente obtenible a partir de n. Sin embargo, esto no se observa en la evidencia; no fue así en los casos de prueba euclídeos, donde quedó en evidencia que el tiempo de ejecución no sigue una relación lineal con respecto a n, y con seguridad tampoco se da esta situación para los casos de prueba de Synopsys.

Tal como indica Helsgaun en el artículo guía de esta memoria [7], el tiempo de ejecución no parece ser un indicador directo que nos permita determinar de antemano el tiempo de ejecución de cada caso.

En particular, estos datos son difícilmente comparables entre sí, ya que solo se pueden obtener conclusiones generales, dado que cada partición se compone de un número altamente variable de cadenas.

4.6. Resultados fundamentales obtenidos

- **Formato:** El algoritmo desarrollado se aplica correctamente tanto a casos euclídeos como asimétricos de TSPLib.
- **Replicación de Resultados:** Se replican correctamente los resultados del artículo original de Helsgaun, con apenas diferencias despreciables. Por tanto, se puede afirmar que este algoritmo obtiene resultados similares a los del artículo original, heredando las propiedades del mismo.
- **Cota Inferior:** Respecto a la cota inferior, el algoritmo obtiene una cota inferior con un error promedio de 1.66 %, medido sobre los casos euclídeos de TSPLib.
- **Optimalidad:** Se obtiene un error de peor caso promedio de 0.04 % para los casos euclídeos simétricos respecto al óptimo real. Para los casos de Synopsys, se obtiene un promedio de optimalidad de 3.15 % con una sola ejecución. Respecto a la optimalidad y el número de ejecuciones, se determina que, para los casos de Synopsys, lo más adecuado es simplemente aplicar el algoritmo con una única ejecución, ya que un aumento de ejecuciones se traduce en una ganancia ínfima, del orden de 0.01 %, mientras que el tiempo aumenta linealmente. Esta decisión se reafirma con la bajísima diferencia de optimalidad obtenida para 1 o 100 ejecuciones sobre TSPLib euclídeo.
- **Tiempo:** En los casos de prueba de Synopsys, el tiempo máximo de ejecución de una partición alcanzó los 1123 segundos (incluyendo Ascent), lo que equivale a 18.7 minutos. En este caso particular, se obtuvo una solución que se sitúa un 3.73 % por encima de la cota inferior. Por otro lado, el tiempo mínimo, junto con una diferencia porcentual entre la solución y la cota inferior del 0.00 %, refleja la simplicidad de los problemas con menor cantidad de vértices. En cuanto al caso promedio para las particiones de Synopsys, se observa un error promedio del 3.15 %, mientras que el tiempo promedio por partición es de 84.55 segundos.

- **Relación con el número de vértices:** Se concluye a partir de los resultados obtenidos en la evaluación que ninguna cantidad, ya sea cota inferior, calidad de la solución y tiempo de ejecución, es obtenible directamente a partir del número de vértices por cadena. Difícilmente se pueden identificar patrones o correlaciones entre la cantidad de vértices y estas cantidades, destacando la importancia de la evaluación de algoritmos en datos de prueba estándar que todos utilicen, para evaluar caso a caso cómo se comporta el algoritmo en comparación con otros, de ahí la importancia de TSPLib.

Capítulo 5

Posible trabajo futuro

Synopsys reconoce la presencia de *scan chains* con más de 100,000 *scan flops*, planteando un desafío significativo en términos de escalabilidad. Este escenario ha impulsado la exploración de nuevas oportunidades para implementar algoritmos capaces de manejar eficientemente problemas de esta magnitud. La búsqueda de soluciones que escalen de manera más efectiva en tiempo se ha convertido en un objetivo fundamental, con el propósito de ofrecer resultados óptimos en un tiempo considerablemente menor.

En la actualidad, el diseño de chips juega un papel crítico en el desarrollo económico global. La creciente demanda de chips mejorados, necesarios para satisfacer las exigencias del mercado, está llevando a un aumento acelerado en la complejidad de estos diseños. Esta tendencia posiblemente resultará en un incremento aún mayor en el número de *scan flops* por *chain* en el futuro. En este contexto, la adaptación y optimización de las tecnologías, incluyendo la gestión eficiente de las *scan chains*, se convierten en aspectos esenciales para abordar los desafíos emergentes en el diseño de chips y garantizar un rendimiento excepcional en medio de un panorama tecnológico en constante evolución.

Ante este desafío, la literatura proporciona una serie en constante desarrollo y actualización de propuestas para abordar eficazmente el problema en cuestión

Optimización de estructuras de datos para TSP: Se puede profundizar en la investigación de estructuras de datos eficientes, especialmente considerando el resultado destacado por [8]. La representación de *tours* para TSP con más de 10^6 vértices utilizando *splay trees* en vez de *two-level trees* podría explorarse en mayor detalle. Esto podría incluir la evaluación de otras estructuras de datos y la adaptación de algoritmos para abordar instancias de TSP de gran escala.

Integración de técnicas de *Machine Learning* en TSP: La tendencia reciente hacia la aplicación de técnicas de *machine learning* para el Problema del Vendedor Viajero (TSP) abre una rica área de exploración. Los avances notables, como el aprendizaje de algoritmos de mejora iterativa utilizando *deep reinforcement Learning* [19, 20], sugieren oportunidades para desarrollar y perfeccionar enfoques basados en *machine learning* para la resolución eficiente del TSP.

Exploración de resultados prometedores: La revisión de resultados altamente pro-

metedores, como los presentados por [21], que destacan por sus sorprendentemente bajos tiempos de ejecución y *trade-off* favorable en optimalidad, proporciona una dirección intrigante para futuras investigaciones. La búsqueda de métodos que logren tiempos de ejecución rápidos sin sacrificar la calidad de la solución es un área clave para el desarrollo de algoritmos eficientes.

Mejora mediante *Supervised Learning*: La propuesta de [22] para mejorar directamente algoritmos implementados mediante la introducción de *supervised learning* en la determinación del conjunto de aristas candidatas abre un espacio interesante para investigar cómo las técnicas de aprendizaje supervisado pueden potenciar y perfeccionar algoritmos existentes en el contexto del TSP.

Uso de versiones más sofisticadas propuestas por Helsgaun: Explorar y evaluar las versiones más sofisticadas propuestas por Helsgaun puede proporcionar perspectivas valiosas sobre posibles mejoras en los algoritmos existentes las cuales sean directamente aplicables sobre el trabajo de esta memoria. Tales versiones se encuentran en [23, 24].

Estas diversas líneas de investigación, entre muchas otras, ofrecen oportunidades emocionantes para expandir el conocimiento y la eficacia en la resolución de problemas complejos del TSP en el proceso de *Reordering* de *scan testing*, proporcionando un amplio espectro de posibles direcciones para futuras investigaciones.

Capítulo 6

Conclusión

Después de una exhaustiva revisión académica donde se caracterizó el problema de *Reordering* y se señaló a TSPLib como el estándar a seguir, se determinó que la clase más prometedora de algoritmos para el desarrollo eran los algoritmos de optimización local. En este contexto, el algoritmo más prometedor resultó ser el algoritmo Lin-Kernighan con las mejoras propuestas por Keld Helsgaun.

La implementación de este algoritmo para Synopsys fue sometida a pruebas intensivas que evaluaron sus tiempos, la calidad de las soluciones y la calidad de su cota inferior. Resolviendo instancias de cientos de vértices en décimas de segundo, obteniendo errores menores al 1 % en TSPLib en sus casos euclídeos y asimétricos y llegando a cotas inferiores en promedio un 1.66 % por encima del óptimo real.

Tras la prueba del algoritmo desarrollado en los casos de prueba de Synopsys, se obtuvieron errores que promediaron alrededor del 3 % por encima de la cota inferior, los cuales conjeturamos son mucho menores con respecto al óptimo real desconocido, resolviendo la mayoría de los casos en apenas unos segundos.

La presente memoria concluye con un nuevo estado del arte del proceso de *Reordering* para Synopsys, la estandarización del proceso basada en la adopción del estándar TSPLib, y la apertura de numerosas oportunidades de aprovechar el conocimiento académico al convertir el problema asimétrico en simétrico. Se ha desarrollado una nueva solución, un algoritmo que supera tanto en optimalidad como tiempo de ejecución en comparación a los algoritmos anteriormente desarrollados por Synopsys.

Mirando hacia el futuro, se presenta un marco integral para que Synopsys continúe desarrollando sus algoritmos de *Reordering*, especialmente orientados a resolver eficientemente cadenas de más de 100.000 vértices.

Se proporcionan directrices claras para que Synopsys continúe desarrollando sus algoritmos basándose en una variedad de soluciones académicas ya existentes o en desarrollo, incluyendo alternativas recientes.

Bibliografía

[1] Williams, T. W. y Parker, K. P., "Design for testability—a survey", Proceedings of the IEEE, vol. 71, no. 1, pp. 98–112, 1983.

[2] Makar, S., "A layout-based approach for ordering scan chain flip-flops", en Proceedings International Test Conference 1998 (IEEE Cat. No. 98CH36270), pp. 341–347, IEEE, 1998.

[3] Girard, P., "Survey of low-power testing of vlsi circuits", IEEE Design & test of computers, vol. 19, no. 3, pp. 82–92, 2002.

[4] Barbagallo, S., Bodoni, M. L., Medina, D., Corno, F., Prinetto, P., y Reorda, M. S., "Scan insertion criteria for low design impact", en Proceedings of 14th VLSI Test Symposium, pp. 26–31, IEEE, 1996.

[5] Lin, K.-H., Chen, C.-S., y Hwang, T.-T., "Layout-driven chaining of scan flip-flops", IEE Proceedings-Computers and Digital Techniques, vol. 143, no. 6, pp. 421–425, 1996.

[6] Mattsson, P., "The asymmetric traveling salesman problem", 2010.

[7] Helsgaun, K., "An effective implementation of the lin–kernighan traveling salesman heuristic", European journal of operational research, vol. 126, no. 1, pp. 106–130, 2000.

[8] Nilsson, C., "Heuristics for the traveling salesman problem", Linkoping University, vol. 38, pp. 00085–9, 2003.

[9] Matai, R., Singh, S. P., y Mittal, M. L., "Traveling salesman problem: an overview of applications, formulations, and solution approaches", Traveling salesman problem, theory and applications, vol. 1, no. 1, pp. 1–25, 2010.

[10] Feuer, M. y Koo, C., "Method for rechaining shift register latches which contain more than one physical book", IBM Technical Disclosure Bulletin, vol. 25, no. 9, pp. 4818–4820, 1983.

[11] Held, M. y Karp, R. M., "The traveling-salesman problem and minimum spanning trees", Operations Research, vol. 18, no. 6, pp. 1138–1162, 1970.

[12] Johnson, D. S., McGeoch, L. A., y Rothberg, E. E., "Asymptotic experimental analysis for the held-karp traveling salesman bound", en Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, vol. 341, p. 350, ACM Press San Francisco, 1996.

[13] Lin, S. y Kernighan, B. W., "An effective heuristic algorithm for the traveling-salesman problem", Operations research, vol. 21, no. 2, pp. 498–516, 1973.

[14] Christofides, N. y Eilon, S., "Algorithms for large-scale travelling salesman problems", Journal of the Operational Research Society, vol. 23, no. 4, pp. 511–518, 1972.

[15] Pop, P. C., Cosma, O., Sabo, C., y Sitar, C. P., "A comprehensive survey on the generalized traveling salesman problem", European Journal of Operational Research, 2023.

[16] Held, M. y Karp, R. M., "The traveling-salesman problem and minimum spanning trees: Part ii", Mathematical programming, vol. 1, no. 1, pp. 6–25, 1971.

[17] Jonker, R. y Volgenant, T., "Transforming asymmetric into symmetric traveling salesman problems", Operations Research Letters, vol. 2, no. 4, pp. 161–163, 1983.

[18] Fredman, M. L., Johnson, D. S., McGeoch, L. A., y Ostheimer, G., "Data structures for traveling salesmen", Journal of Algorithms, vol. 18, no. 3, pp. 432–479, 1995.

[19] Mele, U. J., Gambardella, L. M., y Montemanni, R., "A new constructive heuristic driven by machine learning for the traveling salesman problem", Algorithms, vol. 14, no. 9, p. 267, 2021.

[20] d O Costa, P. R., Rhuggenaath, J., Zhang, Y., y Akcay, A., "Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning", en Asian conference on machine learning, pp. 465–480, PMLR, 2020.

[21] Pan, X., Jin, Y., Ding, Y., Feng, M., Zhao, L., Song, L., y Bian, J., "H-tsp: Hierarchically solving the large-scale travelling salesman problem", arXiv preprint arXiv:2304.09395, 2023.

[22] Xin, L., Song, W., Cao, Z., y Zhang, J., "Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem", Advances in Neural Information Processing Systems, vol. 34, pp. 7472–7483, 2021.

[23] Helsgaun, K., An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic. PhD thesis, Roskilde University. Department of Computer Science, 2006.

[24] Helsgaun, K., "General k-opt submoves for the lin–kernighan tsp heuristic", Mathematical Programming Computation, vol. 1, pp. 119–163, 2009.