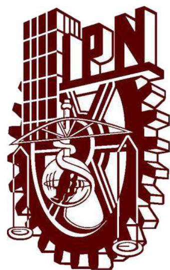


INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN Y DESARROLLO
DE TECNOLOGÍA DIGITAL

MAESTRÍA EN CIENCIAS EN SISTEMAS DIGITALES

“SOLUCIÓN AL PROBLEMA DEL AGENTE VIAJERO APLICANDO ALGORITMOS GENÉTICOS EN PROCESADORES GRÁFICOS”

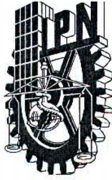
TESIS

QUE PARA OBTENER EL GRADO DE
MAESTRÍA EN CIENCIAS

PRESENTA
ING. FATSIN ERNESTO COTA COTA

BAJO LA DIRECCIÓN DE
DR. JUAN JOSÉ TAPIA ARMENTA

DICIEMBRE 2013 **TIJUANA, B.C., MÉXICO**



SIP-14

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de Tijuana, B.C. siendo las 14:00 horas del día 5 del mes de diciembre del 2013 se reunieron los miembros de la Comisión Revisora de Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de CITEDI para examinar la tesis titulada:

SOLUCIÓN AL PROBLEMA DEL AGENTE VIAJERO APLICANDO ALGORITMOS GENÉTICOS EN PROCESADORES GRÁFICOS.

Presentada por el alumno:

COTA	COTA	FATSIN ERNESTO
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

A	1	2	0	6	2	1

aspirante de:

MAESTRÍA EN CIENCIAS EN SISTEMAS DIGITALES

Después de intercambiar opiniones, los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Director(a) de tesis

DR. JUAN JOSÉ TAPIA ARMENTA

DR. JULIO CÉSAR ROLÓN GARRIDO

DR. ROBERTO SEPÚLVEDA CRUZ

DR. OSCAR HUMBERTO MONTIEL ROSS

M.C. ISAURA GONZÁLEZ RUBIO ACOSTA

PRESIDENTE DEL COLEGIO DE PROFESORES

DRA. MIREYA SARAÍ GARCÍA VÁZQUEZ

S. E. P.
INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN Y DESARROLLO DE TECNOLOGÍA DIGITAL
DIRECCIÓN

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de Tijuana, Baja California, el día **5** del mes **DICIEMBRE** del año **2013**, el (la) que suscribe **FATSIN ERNESTO COTA COTA** alumno (a) del Programa de MAESTRÍA EN CIENCIAS EN SISTEMAS DIGITALES con número de registro **A120621**, adscrito al CENTRO DE INVESTIGACIÓN Y DESARROLLO DE TECNOLOGÍA DIGITAL, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de **DR. JUAN JOSÉ TAPIA ARMENTA**, cede los derechos del trabajo titulado **SOLUCIÓN AL PROBLEMA DEL AGENTE VIAJERO APLICANDO ALGORITMOS GENÉTICOS EN PROCESADORES GRÁFICOS**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección Av. Instituto Politécnico Nacional No. 1310, Mesa de Otay, Tijuana, Baja California, México, C. P. 22510 o a la dirección electrónica: **fatsin.cota.cota@gmail.com o posgrado@citedi.mx.** Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Cota Cota Fatsin E.

FATSIN ERNESTO COTA COTA

Nombre y firma

Dedicatoria

A DIOS por darme la oportunidad de vivir y de tener una familia maravillosa.

Con todo mi amor y cariño para las personas que hacen todo en la vida para que yo pueda lograr mis sueños, por motivarme y guiar mi camino con sus consejos, ejemplo y amor, a ustedes por siempre mi corazón y mi agradecimiento, Papá y Mamá.

A tu paciencia y comprensión, has sacrificado tu tiempo para que yo pudiera cumplir con el mío. Por tu bondad y sacrificio me inspiraste a ser mejor para tí, gracias por estar siempre a mi lado, te amo Gaby, mi Kishita.

A mis hermanas, Lizzeth por haberte convertido desde pequeña en un ejemplo a seguir por el empeño que pones para lograr tus propósitos, Edna que con tus enojos y cariños, me has enseñado que a veces la vida es de contrastes y que debemos buscar el lado bueno de las cosas, salir adelante pase lo que pase. A mi hermano Kelvin por haberte convertido en mi amigo y confidente, por darme tu apoyo y la oportunidad de disfrutar junto contigo los buenos y malos ratos. Sabes lo que significan estas palabras, te quiero así tal como eres, el hermano pequeño, al que quiero y extraño. A los tres los adoro.

Agradecimientos

Agradezco al Instituto Politécnico Nacional, al Centro de Investigación y Desarrollo de Tecnología Digital y al Consejo Nacional de Ciencia y Tecnología por su apoyo para desarrollar y concluir este trabajo de investigación.

Mi más sincero agradecimiento al Dr. Juan José Tapia Armenta, mi director de tesis, por haberme apoyado y orientado en mi investigación, por su valioso tiempo y profesionalismo. De todo corazón, gracias "profe Tapia". También mi agradecimiento a los miembros del comité tutorial, Dr. Julio César Rolón Garrido, Dr. Oscar Humberto Montiel Ross, Dr. Roberto Sepúlveda Cruz y M.C. Isaura González Rubio Acosta por brindarme su apoyo y contribuir con su experiencia a la mejora de este trabajo de investigación.

A mis padres, el Prof. Ernesto Cota y la Sra. Mary Cruz Cota por su apoyo incondicional, por impulsarme a ser mejor persona día a día. Por enseñarme que los propósitos se logran con esfuerzo y trabajo. Gracias padres por darme lo mejor de ustedes y por creer en mí. Gracias por enseñarme a ser feliz.

A mis hermanos, por estar conmigo y apoyarme en todo momento, gracias por ser parte de mi vida, los quiero mucho.

A mi novia y futura esposa, Gaby, por tu apoyo y paciencia, por contribuir a mi crecimiento personal y profesional durante tantos años. Por lo que me has enseñado, por lo que hemos vivido, por lo que hemos compartido, eres el amor de mi vida. Gracias por tu amor mi niña.

A toda mi familia que siempre ha estado pendiente de mí y que me han alentado a seguir siempre adelante.

A mis amigos, quienes están conmigo en buenas y malas, mil gracias por todos los momentos que hemos pasado juntos. A mis compañeros y amigos de CITEDI, por las vivencias que compartimos, y por hacer de esta estadía, una etapa maravillosa de mi vida.

A todos y cada uno de los profesores que en pequeña o gran medida han contribuido en mi desarrollo profesional.

Solución al Problema del Agente Viajero aplicando Algoritmos Genéticos en Procesadores Gráficos

Resumen

En este trabajo se presenta la implementación de Algoritmos Genéticos Paralelos en procesadores multinúcleo y en procesadores gráficos (GPU) para resolver el Problema del Agente Viajero. Se propone un algoritmo basado en el modelo de islas de Algoritmos Genéticos Paralelos. La característica principal del algoritmo es que no se requiere migrar individuos entre las islas, en su lugar se crea una isla Élite con el mejor individuo de cada una de ellas. Al final de cada generación, el mejor individuo de cada isla se escribe en la isla Élite para que el resto de la islas puedan usarlo cuando se aplica el operador de cruce.

El algoritmo propuesto se implementa en un procesador multinúcleo y en una GPU. En la implementación en procesadores multinúcleo se crean varios hilos de ejecución donde cada uno de ellos crea una población de individuos que conforman una isla y los hace evolucionar a través de los operadores genéticos. En cada generación se escribe el mejor individuo en la isla Élite. Al ejecutar el algoritmo, existe la posibilidad de que algunos de los individuos de la población, no sean utilizados para mejorar la población. Esto se debe a que, por ser un proceso aleatorio, posiblemente un individuo nunca sea seleccionado para combinarse con algún otro.

En el Algoritmo Genético Paralelo implementado en GPU, cada bloque de hilos representa una población de individuos, donde cada hilo se encarga de ejecutar los operadores genéticos para cada individuo. Con esto, todos los individuos participan en la evolución de la población, todos son mejorados y por consecuencia la calidad de la población final es mejor.

Los algoritmos fueron implementados para diferentes instancias del Problema del Agente Viajero que fueron tomadas de la biblioteca TSPLIB. Los resultados muestran que la implementación en GPU se acerca más al óptimo reportado en dicha biblioteca.

Palabras Clave: Algoritmos Genéticos Paralelos, GPU, CUDA, Problema del Agente Viajero.

Solving the Traveling Salesman Problem using Genetic Algorithms on Graphics Processors

Abstract

In this paper the implementation of Parallel Genetic Algorithms is presented in multicore processors and graphics processors (GPU) to solve the Traveling Salesman Problem. Based on the island model of Parallel Genetic Algorithms algorithm is proposed. The main feature of the algorithm is that it requires no individuals migrate between islands instead an island Elite with the best individual of each is created. At the end of each generation the best individual of each island is written to Elite Island for the rest of the islands can be applied when the crossover operator is applied.

The proposed algorithm is implemented in a multicore processor and a GPU. In the implementation on multicore processors several threads where each creates a population of individuals that make up an island and evolves through genetic operators are created. When a better solution is found, the wire itself sending a copy of the best individual to Elite Island. When executing the algorithm, there is the possibility that some individuals in the population, are not used to improve the population. This is because, as a random process, possibly an individual will never be selected to be combined with any other.

In the Parallel Genetic Algorithm implemented on GPU, each thread block represents a population of individuals, where each thread is responsible for implementing the genetic operators for each individual. With this, all individuals involved in the evolution of the population are all improved and consequently the quality of the final population is better.

The algorithms were implemented for different instances of the Traveling Salesman Problem that were taken from the TSPLIB library. The results show that the GPU implementation is closer to the optimum reported in that library.

Keywords: Genetic parallel algorithms, GPU, CUDA, Travelling Salesman Problem.

Índice general

Resumen I
Abstract II
Índice General III
Lista de Figuras IV
Lista de tablas VI
Lista de Acrónimos VII

1. Introducción **1**
1.1. Trabajos relacionados 4
1.2. Motivación 6
1.3. Objetivos de la investigación 6
1.3.1. Objetivo general 6
1.3.2. Objetivos específicos 6
1.4. Aportaciones 7
1.5. Organización del contenido por capítulo 7

2. Problema del Agente Viajero **8**
2.1. Formulación del PAV 9
2.2. Definición Formal del PAV 10
2.3. Tipos de PAV 10
2.4. Complejidad del PAV 15

3. Algoritmos Genéticos **16**
3.1. Introducción 16
3.2. Algoritmo Genético Simple 17
3.2.1. Población inicial 18
3.2.2. Función de evaluación 19
3.2.3. Selección 20
3.2.4. Operador de Cruce 21
3.2.5. Operador de Mutación 22

3.2.6. Reemplazo de la población . . . 23
3.2.7. Condición de terminación . . . 23
3.3. Algoritmos Genéticos Paralelos . . . 23
3.3.1. Maestro Esclavo . . . 24
3.3.2. Modelo de grano fino o celular . . . 26
3.3.3. Modelo de grano grueso o de islas . . . 26
3.3.4. Modelos Híbridos o Jerárquicos . . . 28
3.4. Algoritmos Genéticos para el PAV . . . 30
3.4.1. Codificación . . . 30
3.4.2. Población inicial . . . 33
3.4.3. Función de aptitud . . . 33
3.4.4. Selección . . . 33
3.4.5. Operadores de cruce . . . 33
3.4.6. Operador de mutación . . . 35

4. Implementación de Algoritmos Genéticos Paralelos 38
4.1. Introducción . . . 38
4.2. Implementación de un Algoritmo Genético Paralelo en un procesador multinúcleo 40
4.3. Algoritmos Genéticos Paralelos en GPU . . . 42
4.3.1. Unidades de Procesamiento Gráfico . . . 43
4.3.2. Arquitectura CUDA . . . 43
4.3.3. Memorias del GPU y transferencia de datos . . . 46
4.4. Paralelización del modelo de islas de AGP en GPU . . . 48

5. Resultados 53
5.1. Solución al Problema del Agente Viajero con un Algoritmo Genético secuencial . 54
5.2. Solución al Problema del Agente Viajero con Algoritmos Genéticos Paralelos en un procesador multinúcleo . . . 55
5.3. Solución al Problema del Agente Viajero con Algoritmos Genéticos Paralelos en un GPU . . . 57

6. Conclusiones y trabajo futuro 63
6.1. Conclusiones . . . 63
6.2. Trabajo futuro . . . 64

Lista de Figuras

2.1. Ejemplo PAV para 4 ciudades. 9
2.2. Ejemplo del PAV con 5 ciudades 11
2.3. Solución al PAV simétrico con costo de 11 unidades 11
2.4. Solución al PAV asimétrico con costo de 11 unidades 12
2.5. Solución al PAV-MAX con costo de 24 unidades 12
2.6. Soluciones para el PAV con cuello de botella con costo de 4 unidades como mayor costo 13
2.7. PAV con datos agrupados. (a) Ejemplo de PAV con datos agrupados, (b) Solución al PAV con datos agrupados 13
2.8. Solución al PAV generalizado 14
2.9. Solución al PAV con múltiples viajantes 14

3.1. Representación de un individuo como cromosoma 19
3.2. Etapa de selección de AG 20
3.3. Cruce de un solo punto 21
3.4. Mutación aleatoria bit a bit 22
3.5. Esquema de AGP maestro esclavo 25
3.6. Esquema de AGP de grano fino 26
3.7. Modelo de islas de Algoritmos Genéticos Paralelos 27
3.8. Modelo híbrido con modelo de grano grueso en el nivel superior y de grano fino en el nivel inferior 28
3.9. Modelo híbrido con modelo de grano grueso en el nivel superior y maestro-esclavo en el nivel inferior 29
3.10. Modelo híbrido con modelo de grano grueso en el nivel superior e inferior 29
3.11. Problema de 5 ciudades y 20 aristas. 30
3.12. Cromosoma con codificación binaria. 30
3.13. Ruta que representa el cromosoma de la Fig. 3.12. 31
3.14. Cromosoma de un recorrido inválido 31
3.15. Recorrido que representa el cromosoma inválido de la Fig. 3.14 31

3.16. Cruce externo clásico de un punto . . . 32
3.17. Cruce por emparejamiento parcial (PMX) . . . 34
3.18. Cruce por orden (OX) . . . 35
3.19. Cruce por ciclos (CX) . . . 35
3.20. Mutación por inversión de un individuo . . . 36
3.21. Mutación por intercambio de un individuo . . . 36
3.22. Mutación por inserción. (a) Una inserción, (b) Dos inserciones. . . . 36

4.1. Funcionamiento del algoritmo Greedy. . . . 39
4.2. AGP con isla Élite . . . 41
4.3. Modelo de programación CUDA . . . 44
4.4. Organización de los hilos en CUDA . . . 45
4.5. Espacios de memoria en un GPU con arquitectura CUDA . . . 47
4.6. Analogía entre algoritmos genéticos por modelo de islas y GPU . . . 49
4.7. Modelo de islas de AGP con isla Élite . . . 50

5.1. Ruta de la mejor solución obtenida para 131 ciudades . . . 58
5.2. Error de las implementaciones en un procesador multinúcleo y en una GPU . . . 59
5.3. Ruta de la mejor solución obtenida para 237 ciudades . . . 60
5.4. Ruta de la mejor solución obtenida para 380 ciudades . . . 61
5.5. Ruta de la mejor solución obtenida para 662 ciudades . . . 61
5.6. Ruta de la mejor solución obtenida para 813 ciudades . . . 62
5.7. Ruta de la mejor solución obtenida para 1083 ciudades . . . 62

Lista de Tablas

5.1. Cruce por orden (OX) y mutación por inversión. 54
5.2. AG secuencial con población inicial de 1000 individuos. 55
5.3. Soluciones obtenidas en un procesador multinúcleo 56
5.4. AG secuencial vs AGPE en un procesador multinúcleo 56
5.5. Desviación estándar de los resultados en un procesador multinúcleo 57
5.6. Soluciones obtenidas para instancias del TSPLIB con el algoritmo implementado en un GPU 58
5.7. Desviación estándar de los resultados en un GPU 58

Lista de acrónimos

AG Algoritmo Genético
(*Genetic Algorithm*)

AGP Algoritmo Genético Paralelo
(*Parallel Genetic Algorithm*)

AGPE Algoritmo Genético Paralelo con isla Élite
(*Parallel Genetic Algorithm with Elite island*)

CPU Unidad central de procesamiento
(*Central Processing Unit*)

CUDA Arquitectura de dispositivos de cómputo unificado
(*Compute Unified Device Architecture*)

CX Cruce por Ciclos
(*Cycle Crossover*)

FEP Programación Evolutiva Rápida
(*Fast Evolutionary Programming*)

GPGPU Computación de propósito general en unidades de procesamiento gráfico
(*General Purpose Computing on Graphics Processing Units*)

GPU Unidad de Procesamiento Gráfico
(*Graphics Processing Unit*)

MPI Interfaz de Paso de Mensajes
(*Message Passing Interface*)

OX Cruce por Orden
(*Order Crossover*)

PAV Problema del Agente Viajero
(*Travelling Salesman Problem*)

PMX Cruce por Emparejamiento Parcial
(*Partially Mapped Crossover*)

POSIX Interfaz Portátil de Sistema Operativo
(*Portable Operating System Interface*)

RAM Memoria de Acceso Aleatorio
(*Random Access Memory*)

SAT Problema de Satisfacibilidad Booleana
(*Boolean Satisfiability Problem*)

SIMD Una Instrucción, Datos Múltiples
(*Single Instruction, Multiple Data*)

SM Multiprocesador de transferencia continua
(*Streaming Multiprocessor*)

SP Procesador escalar
(*Scalar Processor*)

TSPLIB Biblioteca de instancias del problema del agente viajero
(*Scalar Processor*)

VRAM Memoria Gráfica de Acceso Aleatorio
(*Video Random Access Memory*)

Capítulo 1

Introducción

Tradicionalmente, los programas informáticos se han desarrollado con programación secuencial. Para resolver un problema, se construye un algoritmo y se implementa como un flujo de instrucciones, que se ejecutan en serie en una unidad central de procesamiento (CPU, del inglés Central Processing Unit).

Desde el inicio de la programación de computadoras y hasta el año 2004, una de las formas principales de aumentar el rendimiento de los procesadores fué agregarles más transistores. Para ello, la reducción del tamaño de los transistores permitió que se pudieran utilizar más cantidad de estos en un procesador. El hecho de que los transistores sean más pequeños, les permite cambiar de estado más rápido y aumentar la frecuencia de reloj. Cuantos más pequeños sean los transistores, un procesador podrá tener una mayor cantidad de ellos e incrementar la frecuencia de reloj. Además, el procesador podrá ejecutar más operaciones al mismo tiempo [1, 2].

Anteriormente, en cada generación de procesadores se aumentaba la frecuencia de reloj para mejorar su rendimiento. Con esto, el mismo programa se ejecutaba más rápido en cada nueva generación de procesadores. Sin embargo, esta forma de acelerar la ejecución de los programas en un procesador, se vió limitada debido a que al aumentar la frecuencia de reloj, aumentan también el calor generado así como el consumo de energía, es decir, hasta cierto límite ya no es posible incrementar la frecuencia de reloj [1, 2].

Otra forma de mejorar el rendimiento de un procesador consiste en utilizar el paralelismo a nivel de instrucción. Esta técnica también empezó a ser inviable debido al aumento del costo de añadir hardware adicional que permita encontrar instrucciones que no dependen unas de otras para poderlas ejecutar en paralelo [1, 2].

Es así como, para aumentar la capacidad de procesado, surgió el diseño de procesadores multinúcleo, en los cuales existen varias unidades de procesamiento en el mismo circuito integrado. Cada núcleo de procesamiento tiene que ser controlado por separado, y el sistema operativo puede asignar diferentes programas de aplicación a los diferentes núcleos para obtener una ejecución en paralelo.

Mediante el uso de técnicas de programación en paralelo, es posible ejecutar un programa de cómputo intensivo en un conjunto de núcleos, con lo cual se reduce el tiempo de ejecución comparado con una ejecución en un solo núcleo [2].

Además de los procesadores multinúcleo, desde el año 2003, gracias a la industria de los videojuegos el aumento de la demanda de gráficos en 3D y alta definición en tiempo real ha convertido a los procesadores gráficos (GPU, del inglés Graphics Processing Unit) en auténticos procesadores, arquitecturas altamente paralelas con enorme capacidad de procesamiento y mayor ancho de banda.

Una gran cantidad de problemas, aparte de los relacionados con el procesamiento de gráficos que es para lo que inicialmente fueron diseñadas, pueden beneficiarse con el uso de GPU, es decir, se puede procesar una mayor cantidad de información en menor tiempo. Esto ha promovido el desarrollo de una técnica llamada GPGPU (del inglés General Purpose Computing on Graphics Processing Units), que consiste en el uso de las GPUs para procesar todo tipo de problemas que puedan ser adaptados a dicha arquitectura [3, 4], especialmente en el campo científico y de simulación.

Un programa secuencial se ejecuta en un único núcleo de procesamiento aunque lo ejecutemos en un procesador multinúcleo. En ese sentido, no se aprovecha de la mejor manera la arquitectura de los procesadores multinúcleo o GPU disponibles actualmente. Es por eso la importancia de la programación en paralelo. Asimismo, los programas en paralelo serán los que a futuro aprovechen las mejoras de rendimiento con cada nueva generación de procesadores.

Durante la ejecución de un programa diseñado en paralelo, se utilizan múltiples elementos de procesamiento para resolver el problema. Esto se logra mediante la división del problema en partes independientes de modo que cada elemento de procesamiento pueda ejecutar una parte del algoritmo de manera simultánea con los otros. Los elementos de procesamiento pueden ser una computadora con procesador multinúcleo, varios ordenadores en red o una GPU.

Actualmente, existen una gran variedad de problemas que requieren de una gran cantidad de procesamiento de cómputo [5, 6]. Uno de estos problemas es el Problema del Agente Viajero (PAV) [7] que pertenece al tipo de problemas NP-completos de acuerdo a la teoría de complejidad computacional [5, 6]. Dado un conjunto de ciudades y las distancias entre ellas, el problema es encontrar la ruta con la menor distancia recorrida visitando cada ciudad una vez.

La dificultad de resolver el Problema del Agente Viajero radica en que al aumentar el número de ciudades, aumenta exponencialmente la cantidad de posibles soluciones que tienen que ser evaluadas. Tratar de buscar la mejor solución con un algoritmo de búsqueda exacto, resulta algo inviable debido al tiempo necesario para evaluar la gran cantidad de posibles soluciones.

Como consecuencia de ello, investigadores en la computación han buscado métodos o técnicas que se puedan aplicar para la solución de este problema en un menor tiempo de ejecución. Una de estas técnicas son los Algoritmos Genéticos (AG), que si bien al igual que otras metaheurísticas

no garantizan la obtención de la solución óptima, son capaces de encontrar soluciones cercanas a las óptimas de manera eficiente.

Los Algoritmos Genéticos fueron propuestos inicialmente por John Holland y los desarrolló en su libro [8] en 1975. Holland propuso a los AG como un método heurístico basados en el proceso genético de los organismos vivos. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza de acuerdo con los principios de la selección natural y la supervivencia de los más fuertes. Por imitación de éste proceso, los AG son capaces de ir creando soluciones para problemas del mundo real.

La principal problemática de los AG, radica en la necesidad de evaluar que tan buenos son los individuos de la población [9]. Asimismo, en el caso particular del problema que en este trabajo se aborda, existe la necesidad de calcular una gran cantidad de datos necesarios durante la ejecución del algoritmo.

La ejecución de problemas de mayor tamaño y obtener una solución más aproximada a la óptima, trae consigo un incremento en el tiempo de ejecución. Por este motivo la paralelización es una alternativa interesante tanto para reducir el tiempo de ejecución como para obtener mejores resultados.

Existen distintos modelos de Algoritmos Genéticos Paralelos [9, 10, 11]. Uno de estos es el llamado modelo de islas, el cual consiste en dividir la población total en varias subpoblaciones (islas) en cada una de las cuales se lleva a cabo un Algoritmo Genético. Cada cierto número de generaciones, se realiza un intercambio de individuos entre las subpoblaciones, proceso al que se le llama migración.

El Algoritmo Genético Paralelo que se propone es llamado Algoritmo Genético Paralelo con isla Élite (AGPE). A diferencia del modelo de islas original [10], este no requiere migrar individuos entre las islas, por lo tanto, no existe comunicación directa entre ellas. La característica principal, es que se tiene una isla a la que se le llama Élite donde se encuentra el mejor individuo de cada una de las islas. Los individuos de cada isla pueden combinarse con los individuos de la isla Élite para crear nuevos individuos, pero no con el resto de las islas.

En este trabajo, se resuelven instancias del Problema del Agente Viajero, que debido a la cantidad de ciudades que se manejan no puede ser resuelto por un algoritmo exacto. Se utilizan Algoritmos Genéticos debido a su capacidad de encontrar soluciones cercanas a la óptima de manera eficiente, y se implementa en paralelo para reducir el tiempo de ejecución.

El hacer la implementación en una GPU nos dá la posibilidad de evaluar una mayor cantidad de soluciones y por lo tanto, existe mayor probabilidad de encontrar soluciones más cercanas a la óptima.

Se resolvió el PAV con un Algoritmo Genético secuencial, otra versión paralela con hilos POSIX y una tercera versión con Algoritmos Genéticos Paralelos en GPU.

1.1. Trabajos relacionados

En esta sección, se presentan algunos trabajos relacionados con el Algoritmo Genético Paralelo en GPU propuesto en esta tesis y se describen brevemente las diferencias que tienen con el presente trabajo.

En [12] se presenta un AG donde se paraleliza el operador de cruce y el operador de mutación. Hacen una comparación de la implementación en una GeForce GTX285 versus un CPU Intel de dos núcleos, y se concluye que la implementación en GPU es 24.2 veces más rápida que la implementación en CPU.

En [13] se propone un AG secuencial para resolver el PAV con datos agrupados y no agrupados. El mismo problema se resuelve en un GPU, con una sola población y solo se utiliza un multiprocesador de GPU. Se hace una comparación entre las dos implementaciones y se concluye que el algoritmo con datos agrupados termina más rápido que con datos dispersos, además, la implementación en GPU también termina más rápido en ambos casos.

En [14] se presenta la implementación de los algoritmos de búsqueda local 2-opt y 3-opt en GPU para resolver el PAV con un número de ciudades entre 100 y 4461. Los resultados muestran que la implementación en GPU es de 3 a 26 veces más rápida comparado con la implementación paralela en CPU con 32 núcleos. Sin embargo, no muestran resultados en cuanto a la calidad de la solución obtenida.

En [15] se propone la implementación de un algoritmo genético con el operador de cruce por orden OX para el problema del agente viajero. Se muestran implementaciones en serie y en paralelo de dicho operador, ambos en GPU, así como una implementación en CPU con un núcleo de procesamiento. Los resultados muestran que el algoritmo genético en GPU con el operador de cruce por orden OX en paralelo, es 101.3 veces más rápido que la versión en CPU. Al igual que en [12] se resuelve el problema para problemas con menos de 500 ciudades.

En [16] se presenta un AGP con modelo de islas para solucionar el problema de asignación cuadrática. La subpoblación de cada isla es evolucionada por un multiprocesador de GPU. Los individuos de las subpoblaciones son combinados a través de la VRAM de GPU. Los resultados muestran que la implementación en GPU es de 3 a 12 veces más rápido comparado con la implementación en un procesador Intel i7 965.

En [17] se propone la paralelización de AG en GPU para resolver tres problemas de optimización: máximo global de una función, la función elipsoidal y la función Rosenbrock. Las soluciones para estos problemas se utilizaron con codificación real y codificación binaria. Los resultados son comparados con el algoritmo secuencial en cuanto a precisión y tiempo. Además se hace un análisis de los efectos de modificar el tamaño de la población, número de hebras, tamaño del problema y la complejidad de diferentes problemas.

En [18] se compara la eficiencia de los Algoritmos Genéticos y la evolución diferencial im-

plementados en CUDA. El problema que se resuelve es el problema de programación de tareas independientes donde cada solución candidata es procesada por varios hilos de GPU. Se reporta que se obtuvieron mejores resultados con la implementación de la evolución diferencial.

En [19] se presenta un AGP en GPU que resuelve el problema de ruteo de vehículos. Mediante la asignación de tareas por cada cromosoma para independientes hilos, el algoritmo puede procesar en paralelo todas las operaciones del AG. Los resultados indican que el algoritmo propuesto puede reducir el tiempo de cálculo del problema, lo cual mejora la eficiencia y eficacia del proceso de toma de decisiones.

En [20] se describe cómo aprovechar al máximo un GPU para resolver problemas de propósito general, y de manera específica, expone de que manera se puede implementar AGP en GPU de manera eficiente así como los retos que esto implica.

En [21] se propone un AGP implementado en CUDA para resolver el problema k-SAT y el problema de la mochila. El algoritmo propuesto es 67 veces más rápido que la versión secuencial del problema correspondiente obteniendo la misma calidad de la solución.

En [22] se presenta un AGP basado en el modelo de islas que resuelve el problema de la mochila en un cluster de GPU. Varios GPU hacen evolucionar una isla. La interface MPI es usada para intercambiar material genético entre las islas aisladas. Se utilizaron 14 GPU y 4 procesadores Intel Xeon con seis núcleos cada uno.

En [23] se propone un algoritmo para resolver el PAV para problemas que van desde 131 ciudades hasta 104815. El algoritmo utiliza el método Lin-Kernighan para la búsqueda local, así como un operador de cruce al que los autores nombran Z-Cross. El algoritmo Lin-Kernighan es el que ha desmostrado resolver de mejor manera el problema del agente viajero actualmente. Los resultados demuestran que el Z-Cross obtiene mejores soluciones y se ejecuta más rápido que los operadores de cruce conocidos como PMX3, GSX2 y ERX6.

En [24] se presenta un AG en combinación con un algoritmo de optimización por colonia de hormigas para resolver el PAV. La combinación de los dos algoritmos ayuda a salir de óptimos locales a los que lleva el AG por si solo. Se resuelven problemas que contienen entre 100 y 3038 ciudades, y se concluye que la combinación de los dos algoritmos proporciona mejores resultados que utilizar cada uno de los algoritmos por separado.

En [25] se propone un AGP combinado con un algoritmo de búsqueda taboo para resolver el PAV en GPU. Los problemas resueltos tienen entre 150 y 318 ciudades. Se hace una comparación entre los resultados en GPU y CPU.

En [26] se hace un estudio sobre que etapas de un AG puede ser procesado en GPU. El problema que se resuelve es el Problema del Agente Viajero. Se hacen comparaciones entre las implementaciones en paralelo e implementaciones en procesadores multinúcleo. Finalmente se concluye que todas las etapas de un Algoritmo Genético pueden ser paralelizadas en GPU.

En [27] se presenta la paralelización de un AG híbrido en un GPU. Todos los pasos del algo-

ritmo excepto la generación de números aleatorios se hacen en GPU. Se hace una comparación entre las versiones GPU y CPU del algoritmo paralelo híbrido y el algoritmo de Programación Evolutiva Rápida (FEP, del inglés Fast Evolutionary Programming) propuesto anteriormente por los mismos autores. Los problemas de prueba son algunas funciones de optimización. Se concluye que el AG híbrido converge más rápido y el tiempo de ejecución es más pequeño que el FEP para todos los problemas de prueba.

1.2. Motivación

La mayoría de los trabajos relacionados con el uso de GPU están enfocados a disminuir el tiempo de ejecución de los algoritmos. En este trabajo, además de aprovechar la ventaja que nos ofrece una GPU para reducir el tiempo de ejecución, se busca también que la calidad de la solución sea buena en el sentido de obtener una solución lo más cercana a la óptima que se reporta en TSPLIB. Con la finalidad de mejorar la calidad de la solución se propone un Algoritmo Genético basado en el modelo de islas. A este algoritmo se le llama Algoritmo Genético Paralelo con isla Élite. Aunado a esto, los trabajos revisados hasta la escritura de la presente tesis, se resuelve el Problema del Agente Viajero en un GPU hasta con una cantidad menor a 500 ciudades. En la presente tesis, se presentan resultados para problemas hasta con 1083 ciudades.

1.3. Objetivos de la investigación

1.3.1. Objetivo general

Implementación de un algoritmo genético paralelo para resolver el problema del agente viajero en procesadores gráficos.

1.3.2. Objetivos específicos

- Implementar la paralelización de algoritmos genéticos en procesadores multinúcleo así como en un procesador gráfico.
- Obtener un algoritmo que de solución al problema del agente viajero cuyas soluciones deben ser lo más cercanas posibles a las soluciones óptimas reportadas en TSPLIB.
- Obtener una mejor eficiencia y un menor tiempo de ejecución del algoritmo gracias al uso de GPU en comparación con procesadores multinúcleo.

1.4. Aportaciones

Las aportaciones más destacadas de esta tesis son:

- **Algoritmo genético paralelo basado en el modelo de islas:** Se diseñó un algoritmo genético paralelo basado en el modelo de islas el cual no requiere migrar individuos entre las islas. En su lugar se propone tener una isla Élite donde permanecen los mejores individuos de cada una de las islas. Cualquier isla puede combinar a sus individuos con algún individuo de la isla Élite con la intención de mejorar a sus propios individuos.

- **Algoritmo genético paralelo en GPU para solucionar el PAV:** Se resolvió con algoritmos genéticos paralelos en GPU el problema del agente viajero con una mayor cantidad de ciudades que el número de ciudades que manejan en los trabajos relacionados encontrados hasta la escritura del presente trabajo de investigación.

- **Utilización de GPU:** Se utiliza el máximo número de hilos que se pueden ejecutar en cada bloque de GPU con la intención de no desperdiciar recursos y que todos los hilos disponibles puedan ayudar en el proceso de evolución. Todos los pasos del algoritmo genéticos son ejecutados por un hilo de ejecución para cada individuo de la isla dentro de GPU lo que permite mantenerlos ocupados el mayor tiempo posible.

1.5. Organización del contenido por capítulo

El resto del documento está organizado de la siguiente manera. El capitulo 2 presenta la descripción del problema del agente viajero, donde se explica de manera detallada su definición, formulación y complejidad. La teoría del algoritmo genético simple y paralelo, así como la aplicación de estos al problema del agente viajero es descrito en el capítulo 3. En el capítulo 4 se presenta como se implementaron los algoritmos diseñados en este trabajo de investigación. Se describe un algoritmo genético secuencial, un algoritmo genético paralelo con isla Élite para ser ejecutado por un procesador multinúcleo y finalmente el mismo algoritmo con isla Élite diseñado para su ejecución en GPU. En este capítulo se añade también la teoría sobre el modelo de programación CUDA para tarjetas de procesamiento gráfico. El capítulo 5 muestra los resultados de la presente investigación. Se muestran las tablas correspondientes a los tiempos de ejecución del algoritmo secuencial y los algoritmos en paralelo donde se puede observar también las mejores y peores soluciones obtenidas para 6 instancias del problema del agente viajero. El capítulo 6 muestra las conclusiones y el trabajo a futuro.

Capítulo 2

Problema del Agente Viajero

El Problema del Agente Viajero (PAV) o TSP (por sus siglas en inglés), es probablemente el problema de optimización combinatoria mas estudiado y se ha convertido en un problema típico para validar y examinar nuevas ideas algorítmicas [12]. Además de ser un problema matemático, es uno de los ejemplos más significativos de la teoría de complejidad computacional, con aplicación directa al campo de la inteligencia artificial, y pertenece al conjunto de problemas llamados NP-completos [28].

El problema trata de un viajero que visita exactamente una vez cada una de N ciudades dadas y retorna a la ciudad inicial. La solución del problema consiste en encontrar la secuencia de ciudades visitadas (el ciclo) que minimice la distancia total que recorre el viajero.

La primera noticia que se tiene del PAV fue en 1831; en Alemania un libro fue publicado y titulado como "Der Handlungsreisende" o bien "El Agente Viajero", donde se hace la pregunta ¿Como debe de ser un agente viajero y que debe hacer para vender mas y ser exitoso en su negocio? Para responder a esta pregunta se investigó un algoritmo de búsqueda en el que se cubriera tantas localidades como fuera posible sin visitar una localidad dos veces [7].

Las aplicaciones mas directas y con mayor frecuencia de este problema se aprecian en el campo de la logística. El que una persona necesite recorrer varios lugares, mover mercancías y vehículos de ciudad en ciudad, se adapta muy bien al PAV. De esta manera, la aplicación directa de este problema dentro del campo de la logística se presenta en planificación de ventas, rutas de turismo, de autobuses escolares, así como en reparto de correo.

Además de las aplicaciones en logística, la aplicación del PAV en procesos industriales ha dado lugar a una importante reducción de costos. La secuencia de tareas y producción de circuitos electrónicos han sido los principales procesos donde se aplica. Otra aplicación importante de este problema, se puede mencionar la creación de cluster de datos. La organización de datos en grupos de elementos con propiedades similares es un problema básico en análisis de datos. Este problema, se ha modelado también tomando como base el PAV [29].

2.1. Formulación del PAV

El PAV puede ser formulado de la siguiente manera: sea P_N la colección de todas las permutaciones π del conjunto $\{1, 2, ..., N\}$, entonces para resolver el PAV, se trata de encontrar en P_N una $\pi = (\pi(1), \pi(2), ..., \pi(N))$ de tal manera que $\left[\sum_{i=1}^{N-1} c_{\pi(i)\pi(i+1)}\right] + c_{\pi(N)\pi(1)}$ sea mínima. La expresión, $\left[\sum_{i=1}^{N-1} c_{\pi(1)\pi(i+1)}\right]$ es la distancia necesaria para recorrer desde la primera ciudad hasta la última y $c_{\pi(N)\pi(1)}$ es el costo o distancia necesaria para ir de la última ciudad a la primera.

En este caso $(\pi(1), \pi(2), ...\pi(N))$ muestra el orden en que las ciudades son visitadas, a partir de la ciudad $\pi(1)$, se visita cada ciudad en orden, y después se regresa directamente a $\pi(1)$ desde la ultima ciudad $\pi(N)$. Por ejemplo, si $N = 5$ y $\pi = (2, 1, 5, 3, 4)$ entonces la ruta correspondiente será $(2, 1, 5, 3, 4, 2)$. Cada desplazamiento cíclico de π también da la misma ruta. Por lo tanto, existen N diferentes permutaciones que representan el mismo recorrido.

Una representación del PAV se ilustra en la Fig. 2.1, donde el conjunto de ciudades está dado por $C = \{c_1, c_2, c_3, c_4\}$, siendo las distancias entre cada par de ciudades $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$ y $d(c_3, c_4) = 3$. El ordenamiento $(c_1, c_2, c_3, c_4, c_1)$ es una posible solución para este caso, con una longitud de 28, mientras que el recorrido $(c_1, c_2, c_4, c_3, c_1)$ con una longitud de 27, corresponde a la mejor solución posible para el ejemplo de la Fig. 2.1.

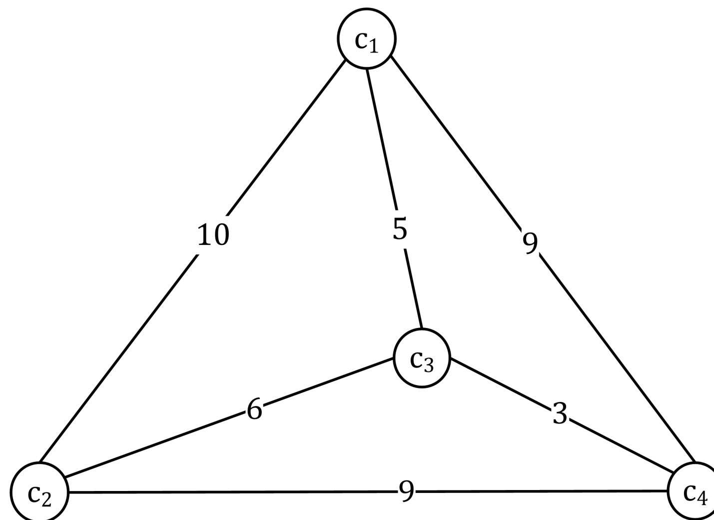


Fig. 2.1: Ejemplo PAV para 4 ciudades.

2.2. Definición Formal del PAV

Muchos problemas de optimización combinatoria pueden ser formulados como problemas de grafos [30, 31]. En este sentido, el Problema del Agente Viajero también puede ser modelado de esta manera. Así, los vértices del grafo representan las ciudades, y los arcos indican los caminos que unen a dichas ciudades. Estos arcos, deben tener un peso, el cual representa la distancia $d(c_i, c_j)$, que hay entre dos vértices conectados por medio de dicho arco. Entonces, una solución de este problema se puede representar como una secuencia de N ciudades, donde una ruta comienza y termina en la misma ciudad.

El PAV modelado como un grafo se presenta a continuación: Sea un grafo $G = (V, E)$ donde $V = \{1, ...N\}$ y $E = \{i, j\} : i, j \in V$ y sea d_{ij} el costo asociado al arco (i, j), además que $X_{i,j}$ sea una variable que nos dice que el viajero va de la ciudad i a la ciudad j ($i = 1, 2..., N$; $j = 1, 2, ..., N$; $i = j$), la función objetivo a minimizar es:

$$\sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} d_{ij} * X_{ij} \tag{2.1}$$

Para garantizar que una ciudad es visitada solo una vez, se tiene:

$$\sum_{i=1}^{N} X_{ij} = 1, (j = 1, 2, 3, ..., N) \tag{2.2}$$

Como se sale una sola vez de cada ciudad, entonces:

$$\sum_{i=1}^{N} X_{ij} = 1, (i = 1, 2, ..., N), \tag{2.3}$$

donde

$$X_{ij} \in \{0, 1\}, (i, j = 1, 2, ..., N) \tag{2.4}$$

2.3. Tipos de PAV

Existe una gran variedad de problemas que son variantes del problema del agente viajero. Se mencionan a continuación los tipos de PAV que tienen mayor similitud con el que se aplica en el presente trabajo de investigación. Para ello, se toma como base el PAV con cinco ciudades mostrado en la Fig. 2.2 y se presenta una solución al mismo considerando las características de cada tipo de PAV explicado.

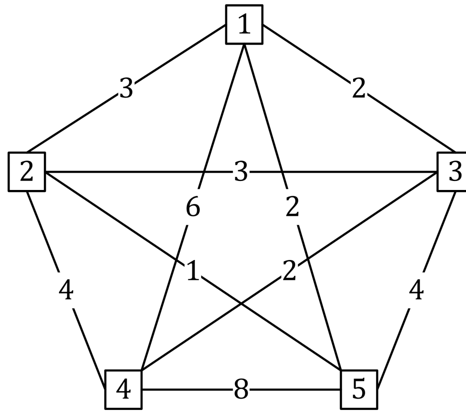


Fig. 2.2: Ejemplo del PAV con 5 ciudades

PAV simétrico. En el PAV simétrico, dado un conjunto de ciudades y distancias entre cada par de ciudades, debe encontrarse el recorrido de menor distancia que visite cada uno de las ciudades exactamente una vez. La distancia recorrida para ir de la ciudad i a la ciudad j es el mismo que ir de j a i. En la Fig. 2.3 se muestra una solución para este tipo de PAV considerando el problema descrito en la Fig. 2.2.

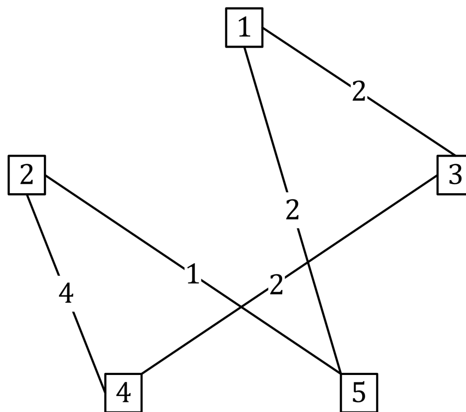


Fig. 2.3: Solución al PAV simétrico con costo de 11 unidades

PAV asimétrico. A diferencia con el anterior, en el PAV asimétrico la distancia que es necesario recorrer para viajar de la ciudad i a la ciudad j es diferente que la distancia necesaria para viajar de la ciudad j a la ciudad i, como se observa en la Fig. 2.4.

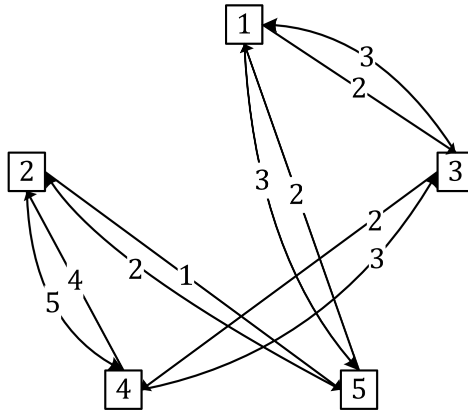


Fig. 2.4: Solución al PAV asimétrico con costo de 11 unidades

PAV geométrico. A este tipo de PAV también se le conoce como Euclidiano. Está dado por un conjunto de puntos $S = \{S_1, S_2, ..., S_n\}$ que corresponden a las ciudades del problema, en un plano con métricas euclidianas, para el que se busca el recorrido mas corto que visite todos los puntos de S, donde las distancia entre dos puntos está dado por la distancia euclidiana entre ellos.

PAV-MAX. Consiste en encontrar una ruta de coste máximo. Este tipo de PAV se presenta en la Fig. 2.5.

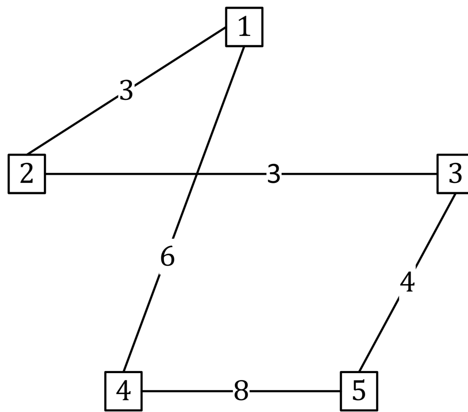


Fig. 2.5: Solución al PAV-MAX con costo de 24 unidades

PAV con cuello de botella. En este problema, se debe encontrar una ruta tal que minimice la mayor distancia de recorrido, en vez de minimizar el coste total. En la Fig. 2.6 se muestran dos soluciones a este tipo de PAV cuyo mayor coste está minimizado a 4 unidades.

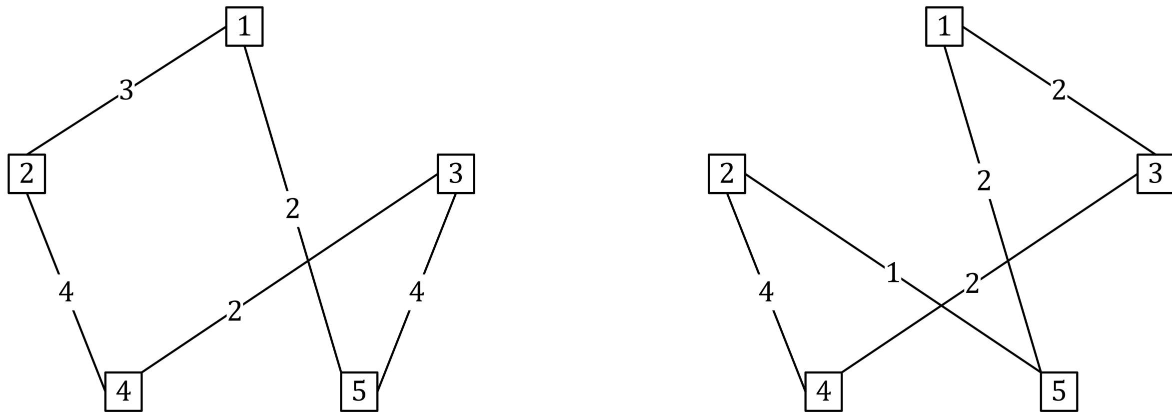


Fig. 2.6: Soluciones para el PAV con cuello de botella con costo de 4 unidades como mayor costo

PAV agrupado. Los nodos o ciudades están divididos en "clusters" o grupos, de manera que lo que se busca es una ruta de coste mínimo en el que se visiten todos los nodos de los grupos. Se deben visitar primero todos los nodos de un grupo para pasar al siguiente.

En la Fig. 2.7(a) se muestra un ejemplo de este tipo y cuya solución se presenta en la Fig. 2.7(b). Cada punto representa una ciudad de destino, y C_1, C_2, C_3, C_4 representan grupos de ciudades.

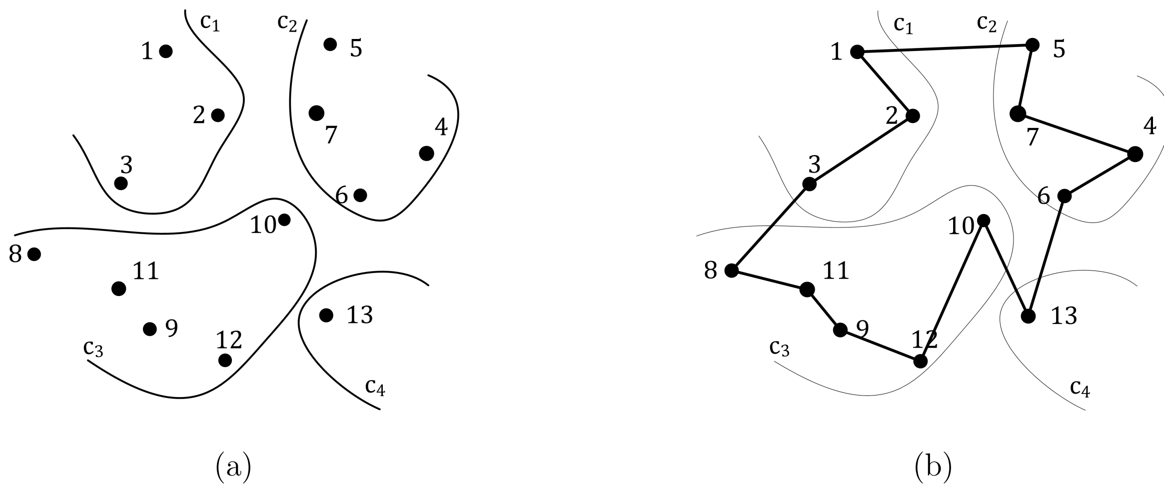


Fig. 2.7: PAV con datos agrupados. (a) Ejemplo de PAV con datos agrupados, (b) Solución al PAV con datos agrupados

PAV generalizado. Los nodos o ciudades también están divididos en grupos, pero lo que se busca es una ruta de coste mínimo que visite exactamente una ciudad de cada grupo. Tomando como ejemplo la Fig. 2.7(a), se muestra en la Fig. 2.8 una solución al PAV generalizado.

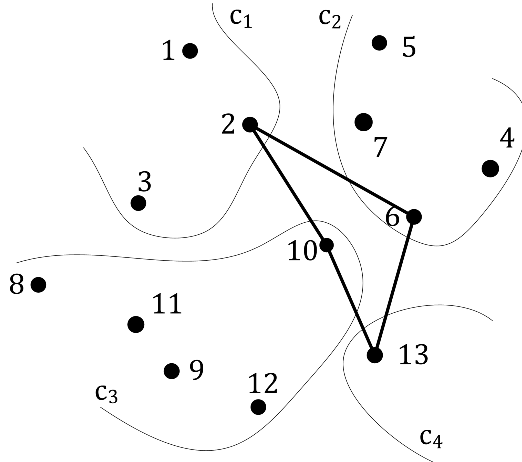


Fig. 2.8: Solución al PAV generalizado

PAV con múltiples viajantes. Existen un número M de viajantes, cada uno de los cuales debe visitar algunas de las ciudades. El problema se transforma en la búsqueda de una partición de los nodos a visitar $X_1, ..., X_M$ y de M rutas, uno para cada X_i, de manera que la suma de las distancias recorridas por los M viajantes sea mínima. En la Fig. 2.9 se ilustra la idea principal de como trabaja el PAV bajo este esquema. Cada ruta inicia y termina en A, y para este caso $M = 3$.

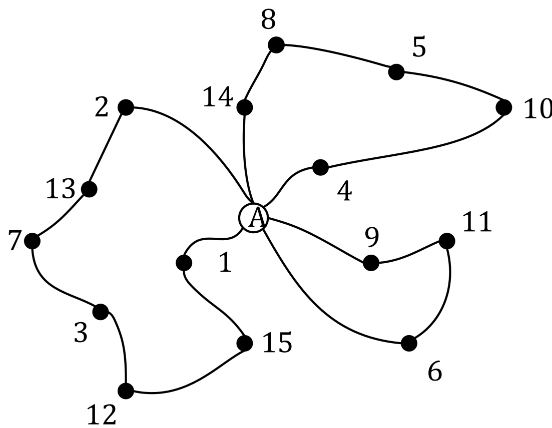


Fig. 2.9: Solución al PAV con múltiples viajantes

El tipo de PAV que se aplica en este estudio, es el denominado geométrico o euclidiano, donde se tiene un conjunto de puntos con sus respectivas coordenadas en un plano. Cada uno de estos puntos representan las ciudades a visitar y la distancia entre cada par de ciudades, está determinada por la distancia euclidiana existente entre ellas. Además, es el denominado PAV simétrico, donde la distancia recorrida para ir de la ciudad i a la ciudad j es la misma que si se hace el recorrido de manera inversa.

2.4. Complejidad del PAV

Para resolver el Problema del Agente Viajero la solución más directa es evaluar todas las posibles combinaciones de rutas y elegir como mejor solución aquella cuya distancia de recorrido sea menor. El problema reside en el número de posibles combinaciones que viene dado por el factorial del número de ciudades. Esto hace que la solución por búsqueda exhaustiva para este problema no sea posible.

En el caso del PAV simétrico el número de rutas posibles y diferentes es $(N-1)!/2$. Existen $(N-1)$ posibilidades a partir de la primer ciudad, $(N-2)$ posibilidades para la siguiente ciudad y así sucesivamente. El denominador 2, surge debido a que si se invierte el sentido de cada ruta, esta cuenta como otra ruta posible pero con la misma distancia.

El PAV pertenece al tipo de problemas NP completos. Este tipo de problemas, son los más difíciles de resolver según la teoría de complejidad computacional [5, 6].

El tiempo necesario para resolver el problema del agente viajero se incrementa a medida que aumenta el número de ciudades N. Por ejemplo, si cada ruta tiene 30 ciudades, en total serían $2{,}65 \times 10^{32}$ sumas. Si se considera que un ordenador es capaz de calcular la longitud de cada ruta en un microsegundo, tardaría algo más de 3 segundos en resolver el problema para 10 ciudades, algo más de medio minuto en resolver el problema para 11 ciudades y 77147 años en resolver el problema para sólo 20 ciudades. Además, el hecho de añadir una ciudad más, provoca que el número de sumas se incremente por un factor de 21. Por lo tanto, para solucionar este problema con esa cantidad de ciudades, son necesarios muchos recursos de cómputo, así como bastante tiempo de ejecución del algoritmo.

El PAV es un problema de optimización combinatoria en el que su espacio de búsqueda crece exponencialmente con el número de ciudades. Para cierto número de ciudades en adelante, no existen métodos de solución eficientes que garanticen encontrar la ruta óptima. A consecuencia de ello, los métodos de solución que se emplean generalmente, encuentran una solución aproximada a la óptima en un tiempo razonable [7].

Capítulo 3

Algoritmos Genéticos

3.1. Introducción

Los Algoritmos Genéticos (AG) propuestos en 1975 por John Holland [8] son técnicas de búsqueda y optimización que se basan en los principios de la evolución biológica propuestos por Darwin, para encontrar la mejor solución a un problema dado entre un conjunto de posibles soluciones.

En la naturaleza, las poblaciones evolucionan generación tras generación de acuerdo a los principios de la selección natural y la supervivencia de los mas fuertes, según la teoría de Darwin. Los individuos de una población compiten entre sí en la búsqueda de recursos para poder sobrevivir. Aquellos con mayor capacidad de sobrevivencia tienen mayor probabilidad de generar un mayor número de descendientes. Por lo tanto, los genes de los individuos mejor adaptados se propagarán en sucesivas generaciones hacia un mayor número de individuos. Así, las especies evolucionan logrando características cada vez mejor adaptadas al entorno en el que viven.

Los Algoritmos Genéticos, evolucionan de manera similar al proceso genético de los organismos vivos. Se inicia con una población de individuos, cada uno de los cuales representa una posible solución factible a un problema dado. A cada individuo se le asigna un valor que representa que tan "buena" es dicha solución con respecto a las demás. En la evolución biológica, esto equivale al grado de efectividad de un organismo para competir por determinados recursos. Mientras mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que sea seleccionado para reproducirse por medio de los operadores de cruce o de mutación. Por el contrario, mientras menor sea la adaptación de un individuo, reduce la probabilidad de que dicho individuo sea seleccionado para la reproducción, y de que su material genético se propague en sucesivas generaciones.

Con la reproducción, se produce una población nueva de posibles soluciones, la cual con-

tiene mejores individuos y reemplaza a la población anterior. De esta manera, las buenas características de la población se propagan generación tras generación. Debido a que se favorecen los individuos mejor adaptados, se van explorando las áreas más prometedoras del espacio de búsqueda.

Los Algoritmos Genéticos se caracterizan por tener un campo de aplicación diverso. Basta con definir una representación del problema y una función de evaluación de los individuos para aplicarlos en la solución de una gran variedad de problemas.

La ejecución del ciclo reproductivo de un AG simple puede requerir una gran cantidad de recursos computacionales (por ejemplo, memoria de gran tamaño y tiempos grandes de búsqueda), por lo tanto, una variedad de cuestiones algorítmicas han sido estudiados para diseñar AG más eficiente. Para mejorar los resultados de los AG, numerosos avances se están logrando mediante el diseño de nuevos operadores y algoritmos híbridos [16, 32], así como también, incluyendo al paralelismo de AG como un método de obtener mejoras con respecto a un AG simple.

Actualmente, existe un gran número de implementaciones de Algoritmos Genéticos Paralelos (AGP) aplicados a una gran variedad de problemas. Las razones de este éxito tienen que ver, en primer lugar, con el hecho de que un AG simple es propenso al paralelismo, ya que la mayoría de los operadores de variación pueden ser fácilmente realizadas en paralelo. Sin embargo, la razón más importante es el uso de una población estructurada, es decir, una distribución espacial de los individuos, en la forma de un conjunto de islas [22] o una rejilla de difusión [33], es responsable de tales beneficios. Por esta razón, muchos autores no utilizan una maquina paralela para ejecutar un AGP y aún así, obtienen mejores resultados que al aplicar un AG simple secuencial [34].

3.2. Algoritmo Genético Simple

La popularidad de los Algoritmo Genéticos ha crecido en forma notoria en los últimos años como consecuencia de su versatilidad para resolver problemas. En términos de Algoritmos Genéticos, a las soluciones posibles del problema dado se le llama individuos de la población. La población está formada por todo el conjunto de individuos, siendo la población inicial el conjunto de soluciones iniciales que son creadas para hacerlas evolucionar con la intención de mejorarlas.

Se le llama operadores genéticos a las acciones que se realizan para evolucionar a las soluciones iniciales, siendo selección, cruce y mutación los más utilizados. Éstos determinan el modo en que el algoritmo explora el espacio de soluciones del problema. Una gran diversidad de propuestas de operadores genéticos han surgido a lo largo de la historia de los Algoritmos Genéticos, siendo los operadores de selección, cruce y mutación así como sus múltiples variantes los más referenciados en la literatura.

Algoritmo 1 Pseudocódigo de un AG simple

```
generar población inicial
evaluar la adaptación de cada individuo
while TERMINO = FALSO do
   seleccionar individuos de la anterior generación
   cruzar individuos seleccionados
   mutar individuos de la población
   evaluar la adaptación de los descendientes obtenidos por cruce y mutación
   insertar en la nueva generación los descendientes obtenidos
end while
terminar
return la mejor solución encontrada
```

El Algoritmo 1, muestra el pseudocódigo de un Algoritmo Genético. Para el buen funcionamiento del algoritmo, es necesario una codificación o representación de las soluciones del problema, que sean adecuadas al mismo. Además, se requiere una función de ajuste o adaptación al problema. Dicha función, permite cuantificar la calidad de la solución representada por el individuo.

Durante la ejecución del algoritmo, los padres deben ser seleccionados para la reproducción. Luego, dichos padres se cruzan generar hijos. Posteriormente, con cierta probabilidad se mutan los individuos de la población. El resultado de la combinación de las anteriores funciones, será un conjunto de individuos (posibles soluciones al problema) que formarán parte de la siguiente población. Ésta debe contener mejores individuos que la población anterior.

3.2.1. Población inicial

La población inicial se forma a partir de las posibles soluciones del problema, donde una posible solución equivale a un individuo en términos de AG. Es así como un conjunto de N individuos representa una población inicial de tamaño N, donde N es un parámetro de entrada del AG y generalmente se conserva durante la ejecución del algoritmo.

En un Algoritmo Genético, los individuos de la población inicial, que son las posibles soluciones del problema, pueden representarse como un conjunto de parámetros conocidos como *genes*. Estos, se agrupan y forman una cadena de valores a la que se le llama *cromosoma*, de tal manera que, cada solución del problema tendrá su correspondiente *cromosoma*, y que a su vez, es un individuo de la población. Los *genes* son representados por una serie de símbolos que generalmente pertenecen a algún sistema numérico [11]: binario, decimal, octal, etc.

La mayoría de los trabajos consultados hasta la escritura del presente trabajo, muestran que la codificación binaria ha sido la más utilizada. Esto se debe a que esta forma de codificación permite la aplicación de operadores genéticos sencillos, además de que las soluciones de la

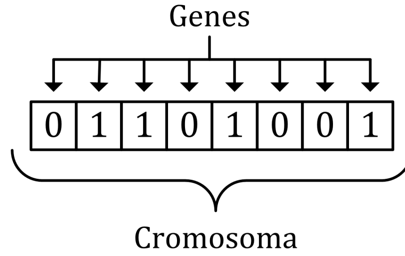


Fig. 3.1: Representación de un individuo como cromosoma

mayoría de los problemas tratados, pueden ser representados por esta codificación. En la Fig. 3.1 se muestra la codificación de un *cromosoma* cuyos *genes* están representados por ceros y unos (sistema binario).

Generalmente, los individuos que forman parte de la población inicial son generados de forma aleatoria, donde cada una de las posibles soluciones al problema, tienen la misma probabilidad de ser parte de la población inicial. Al hacer esto aleatoriamente, se obtiene una población inicial uniformemente distribuida en el espacio de búsqueda, lo cual favorece la exploración. Sin embargo, la calidad media de las soluciones generadas de esta forma será baja, como se muestra en los resultados experimentales de este trabajo, necesitando más iteraciones o generaciones para ser mejoradas.

Otra forma de inicializar la población, es utilizar una o varias heurísticas. Los individuos generados con estas técnicas resultan ser mejores que las generadas de forma aleatoria. Sin embargo, posiblemente vayan dirigidas a zonas determinadas del espacio de búsqueda, con lo que se pierde diversidad en el proceso. No obstante, el partir de soluciones ya mejoradas, reducirá el tiempo de ejecución del AG.

La capacidad de exploración inicial depende del número total de posibles soluciones y el número de individuos que forman parte de la población. Cuanto mayor sea el número de individuos de la población mayor será el poder exploratorio del algoritmo. Sin embargo, al incrementar los individuos de la población aumenta el tiempo de cómputo.

3.2.2. Función de evaluación

La evolución de la población, depende de la calidad relativa de los individuos que compiten por aumentar su presencia en la población y por participar en las operaciones de reproducción. En un problema de búsqueda u optimización, dicha calidad se mide por la adecuación o adaptación de cada individuo para ser solución del problema y se le llama *aptitud* [35]. Ésta, es simplemente un valor numérico que se calcula mediante una función de evaluación específica del problema, a la cual se le denomina *función de aptitud*. Entonces, la *aptitud* de un individuo, es un número real que refleja el nivel de adaptación al problema.

La *función de aptitud* tiene una influencia importante en el funcionamiento del AG, ya que guía el mecanismo de exploración en el espacio de búsqueda. Además, debe contemplar el criterio del problema de optimización ya sea minimización o maximización de un objetivo, así como las restricciones presentes en el mismo.

3.2.3. Selección

Una parte fundamental del funcionamiento de un algoritmo genético es el proceso de selección de candidatos a reproducirse. La selección es el proceso mediante el cual individuos de la población son elegidos para reproducirse y crear a los individuos que formarán la siguiente generación. A los individuos seleccionados se les llama padres.

La selección de padres se efectúa al azar usando algún procedimiento que favorezca a los individuos con mejor valor de aptitud. Sin embargo, es necesario también incluir un factor aleatorio que permita reproducirse a individuos que aunque su valor de aptitud no sea muy buena, puedan contener alguna información útil para posteriores generaciones, con el objetivo de mantener así también una cierta diversidad en cada población.

Existen muchas técnicas de selección usadas en Algoritmos Genéticos, entre las cuales destacan selección por ruleta, por torneo y selección de Boltzman. Su funcionamiento, así como otras técnicas de selección, pueden ser consultadas en [9, 11]. Independientemente del procedimiento que se utilice para hacer la selección de padres, todos se rigen por la evaluación de la función aptitud. Los individuos con mejor aptitud tienen más posibilidad de ser seleccionados.

En la Fig. 3.2 se muestra el proceso de selección. De la población se seleccionan los individuos con mejor aptitud, para aplicarles el operador de cruce y obtener nuevos individuos.

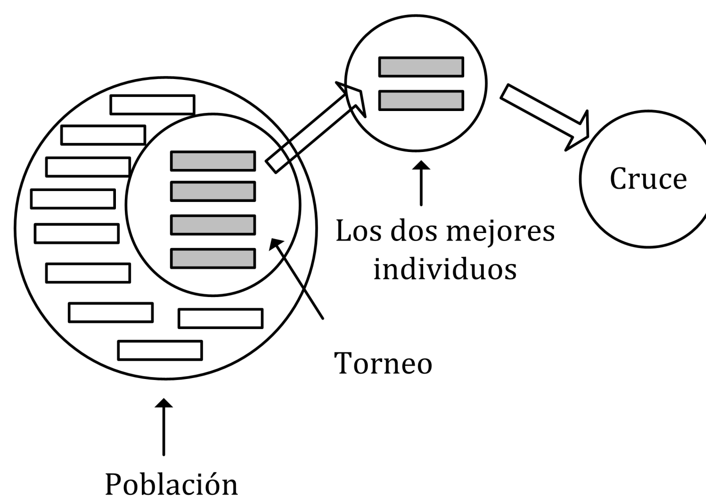


Fig. 3.2: Etapa de selección de AG

3.2.4. Operador de Cruce

El operador de cruce, permite generar individuos nuevos que heredan características de los padres. El cruce, dentro del AG es manejado mediante el porcentaje de cruce P_c, el cual indica si se va a efectuar la recombinación o no, esto quiere decir, que habrá algunos individuos que pasarán a la siguiente generación, directamente sin cruzarse [9, 10, 35]. No existe un valor único de P_c que asegure el buen desempeño del AG. El mejor valor de P_c está relacionado con los valores del resto de los parámetros del algoritmo. Al modificar dichos parámetros, es necesario ajustar el valor de P_c para el buen funcionamiento del algoritmo.

Los individuos seleccionados son recombinados para producir los hijos que formarán parte de la siguiente generación. Existen ciertos criterios para que los hijos pasen a ser parte de la siguiente generación. Pueden ser insertados en la población aunque sus padres tengan mejor aptitud, ó que pasen a la siguiente generación únicamente si su aptitud es mayor que la de los padres o de los individuos a reemplazar.

La idea del cruce, se basa en que si se toman dos individuos y se obtiene una descendencia que comparta genes de ambos, existe la posibilidad de que los genes heredados sean los que ayudan a los hijos a tener buena aptitud. Al compartir las características buenas de dos individuos, la descendencia, o al menos parte de ella, debería tener una mejor aptitud que cada uno de los padres por separado. Si el cruce no agrupa las mejores características en uno de los hijos y estos tienen una peor aptitud, si así se elige, se pueden insertar a la población y ayudar así a la diversidad de búsqueda.

Al igual que en la selección, existen una gran cantidad de técnicas de cruce [9, 11]. Tienen diferentes grados de complejidad y de igual manera, unas pueden adaptarse mejor que otras a ciertos problemas de aplicación. En la Fig. 3.3 se muestra el cruce de un punto. La imagen muestra individuos con codificación binaria.

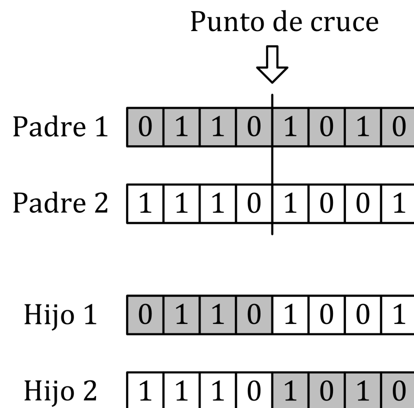


Fig. 3.3: Cruce de un solo punto

3.2.5. Operador de Mutación

La mutación se considera un operador básico que proporciona un pequeño elemento de aleatoriedad a los individuos de la población. Si bien se admite que el operador de cruce es el responsable de efectuar la búsqueda a lo largo del espacio de posibles soluciones, el operador de mutación va ganando importancia a medida que la población de individuos va convergiendo. Esto se debe a que al ir convergiendo el AG, los individuos son cada vez mas parecidos entre sí. Entonces, aplicar el operador de mutación permite modificar los individuos con la intención de ganar diversidad de búsqueda.

El objetivo del operador de mutación es producir nuevas soluciones a partir de la modificación de un cierto número de genes de una solución existente representada por un individuo, con la intención de fomentar la variabilidad dentro de la población.

La mutación puede ser aplicada a todos o algunos miembros de la población. La selección del gen o los genes a modificar se realiza con ayuda de cualquier método que emplee una técnica de selección aleatoria.

Se puede seleccionar los individuos directamente de la población actual y mutarlos antes de instroducirlos en la nueva población, ó aplicar la mutación directamente a los hijos después de haberlos obtenido por cruce.

Al igual que el operador de cruce, el operador de mutación se emplea con cierta probabilidad, que en este caso se denomina P_m. Ésta indica si la mutación se hace o no y normalmente es pequeña [9].

Existen diferentes formas de realizar mutación [11], la más fácil es la llamada mutación aleatoria bit a bit que se aplica al tipo de codificación binario [35]. En la Fig. 3.4 se muestra este tipo de mutación.

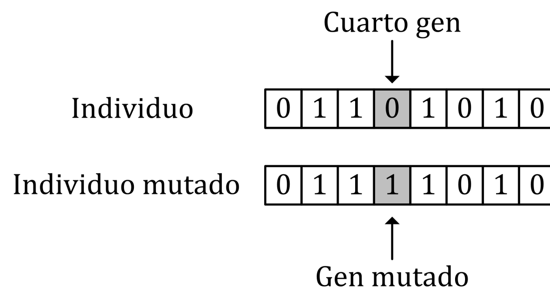


Fig. 3.4: Mutación aleatoria bit a bit

3.2.6. Reemplazo de la población

Cada vez que se aplica el operador de cruce y/o mutación, nos encontramos con un número determinados de nuevos individuos (la descendencia) que se han de integrar en la población para formar la siguiente generación. A esta operación se le denomina reemplazo y se puede hacer de diversas formas [9, 36]. A continuación se mencionan algunas de ellas y que tradicionalmente son las más utilizadas.

- Cuando el número de individuos nuevos llega a un cierto valor, se elimina un subconjunto de la población. Este subconjunto contiene a los individuos peor adaptados.
- Cada vez que se crea un individuo nuevo, en la población se elimina el peor adaptado para dejar su lugar a este.
- Cada vez que se crea un individuo nuevo, en la población se elimina aleatoriamente una solución, independientemente de su adaptación
- Cuando se crea un individuo, se elimina al padre o a uno de los padres en caso de que el descendiente tenga mejor aptitud.

3.2.7. Condición de terminación

La evolución de las soluciones debe terminar en algún momento, es por eso que se debe establecer en que condición el algoritmo terminará de ejecutarse. A esta condición o criterio se le llama condición de terminación.

La condición de terminación, generalmente viene determinada por criterios sencillos, como un número máximo de generaciones o un tiempo máximo de resolución. También, suele determinarse más eficientemente por estrategias relacionadas con indicadores del estado de evolución de la población, así como por la pérdida de diversidad dentro de la población o por no haber mejora en un cierto número de iteraciones. Por lo general una condición mixta es lo más utilizado, es decir, limitar el tiempo de ejecución a un número de iteraciones y tener en cuenta algún indicador del estado de la población para considerar la convergencia antes de alcanzar tal limitación. Si se supiera la respuesta a la que se debe llegar, se podría detener el Algoritmo cuando se obtenga tal solución o utilizar algún criterio de error. Sin embargo, eso casi nunca es posible.

3.3. Algoritmos Genéticos Paralelos

Una de las principales ventajas de los Algoritmos Genéticos es que permite que sus operaciones se puedan ejecutar en paralelo. Debido a que la evolución natural trata con una población

entera y no con individuos particulares, las operaciones del AG pueden ser procesadas de manera independiente para cada uno de los individuos.

Los Algoritmos Genéticos Paralelos (AGP) surgen ante la necesidad de cómputo requerida por problemas de extrema complejidad, cuyo tiempo de ejecución utilizando los Algoritmos Genéticos secuenciales es demasiado grande. Además de reducir el tiempo de ejecución, los AGP permiten acercarse más a la solución óptima. Gracias al paralelismo, es posible incrementar el tamaño de la población, reducir el costo computacional y mejorar el desempeño de los AG.

La adaptación de este tipo de heurísticas a distintas configuraciones de cómputo paralelo, dió lugar a varios modelos de AGPs: algoritmos maestro-esclavo, algoritmos de grano fino y algoritmos de grano grueso o de islas, así como también los llamados algoritmos genéticos paralelos híbridos [9, 10, 11, 37].

En Algoritmos Genéticos, se puede hacer paralelismo de tal manera que un algoritmo se ejecute muchas veces en forma concurrente, y además, las operaciones dentro del mismo, pueden también hacerse en paralelo. Por ejemplo, al momento de crear a los individuos de la población inicial, se pueden crear de manera simultanea todos los individuos. De la misma manera, la etapa de evaluación, y los operadores cruce y mutación, pueden hacerse para cada individuo o para cada grupo de individuos respectivamente de forma independiente.

Como se muestra enseguida, existen algunos modelos de algoritmos genéticos paralelos que utilizan un operador genéticos llamado migración. En este caso, esta operación también puede ejecutarse en paralelo ya que no existe dependencia de datos entre los individuos que se emigran de una subpoblación a otra.

La implementación de la paralelización de los operadores genéticos antes mencionados, reduce el tiempo de ejecución del algoritmo y por lo tanto, nos permite aumentar el numero de soluciones a evaluar donde existe mayor posibilidad de encontrar una mejor solución.

3.3.1. Maestro Esclavo

Probablemente la forma mas fácil de implementar un AGP es hacer la etapa de evaluación de la aptitud de los individuos entre varios procesos esclavos, mientras un proceso maestro ejecuta el resto de las operaciones del AG (selección, cruce y mutación). Estos algoritmos exploran el espacio de búsqueda de la misma forma que un AG en serie y en algunos casos se obtiene una significante mejora en el rendimiento [37].

Estos algoritmos trabajan con una única población de individuos que será gestionada por el nodo maestro. La evaluación de costo de los individuos y la aplicación de los operadores genéticos se realiza por los nodos esclavos. A cada nodo esclavo le corresponderá una parte de la población total, sobre la cual realizará las operaciones antes citadas. Una vez terminado este proceso, devolverán el resultado al nodo maestro, que realizará la selección de individuos. En la

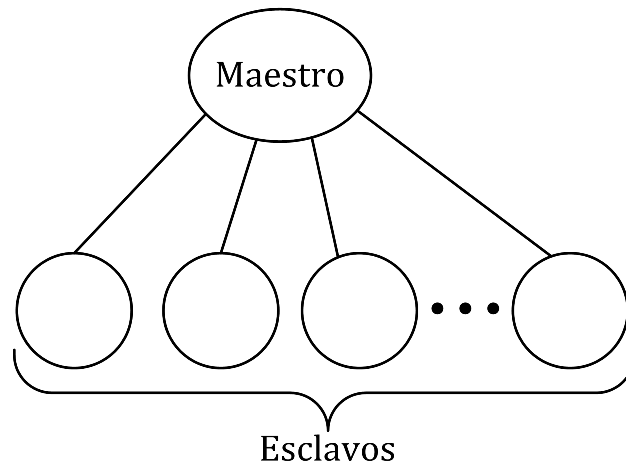


Fig. 3.5: Esquema de AGP maestro esclavo

Fig. 3.5 se muestra un esquema de la arquitectura de este tipo de algoritmos.

Normalmente, la operación que se suele implementar en paralelo es la evaluación de la aptitud de los individuos, porque suele ser la más compleja. Además, este valor es independiente del resto de la población, por lo que su implementación en paralelo es muy sencilla. La comunicación se dá entre el nodo maestro y los esclavos. El nodo maestro envía el subconjunto de individuos que corresponde a cada nodo esclavo, y éstos le devuelven los valores de costo para cada uno de ellos. La comunicación puede ser implementada de dos maneras: síncrona o asíncrona. En la primera de ellas, el nodo maestro espera a recibir los valores de costo de todos los individuos para generar la siguiente generación. En la segunda, el algoritmo no espera a que los nodos más lentos envíen sus valores de costo. De este modo conseguimos agilizar el proceso, pero el comportamiento no es exactamente el mismo que el de un algoritmo genético secuencial. La implementación síncrona, en cambio, sí mantiene este comportamiento.

Este tipo de algoritmos pueden ser implementados sin problemas en computadores tanto de memoria compartida como de memoria distribuida. En el caso de los computadores de memoria compartida, la población puede estar almacenada en memoria, y cada procesador esclavo puede leer directamente los individuos que le sean asignados. En caso de usar un computador de memoria distribuida, sería el nodo maestro el encargado de asignar y distribuir individuos entre los nodos esclavos y de recoger, una vez evaluada, la adecuación de cada uno de ellos.

Una característica importante de estos algoritmos es que los resultados son los mismos que los de la ejecución secuencial, solo cambia en tiempo de ejecución, ya que el nodo maestro ejecuta un AG simple [35].

3.3.2. Modelo de grano fino o celular

Estos modelos trabajan con una única población estructurada espacialmente. Han sido diseñados para ser implementados en arquitecturas de memoria compartida masivamente paralelas, aunque también existen trabajos que demuestran su eficiencia en arquitecturas paralelas distribuidas [33, 38].

En este modelo la población se divide de tal manera que cada procesador albergue un individuo. La selección y el cruce de individuos se hará entre individuos que pertenezcan a un mismo vecindario, formado por un conjunto de individuos adyacentes según la representación espacial antes citada. Sin embargo, se permite el traslape entre vecindarios, lo que permite que cada individuo forme parte de varias subpoblaciones, y con esto propiciar la interacción entre todos los individuos de la población.

La selección y la reproducción de cada individuo, solo se puede hacer con sus vecinos. Como las vecindades se solapan, se produce una tendencia a que los mejores individuos se expandan por toda la población. En la Fig. 3.6 se observa un esquema de este modelo.

La proximidad de individuos está determinada por la topología de interconexión entre los elementos de procesamiento. La alta conectividad entre vecinos incrementa la difusión de los individuos con mejor aptitud.

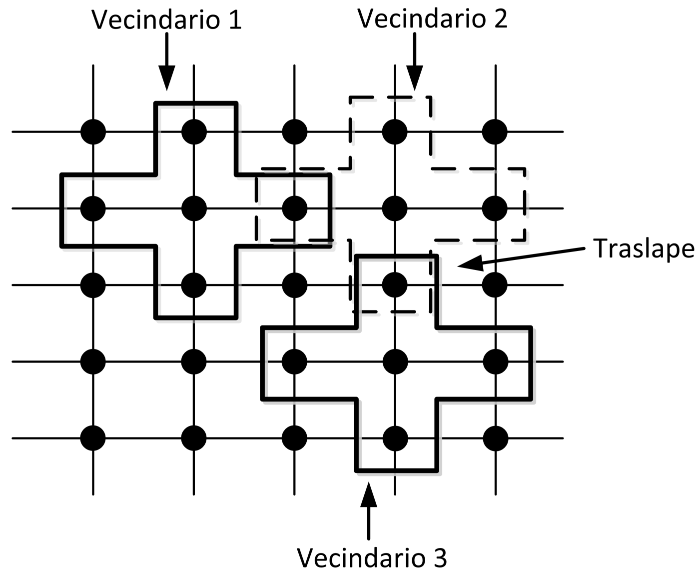


Fig. 3.6: Esquema de AGP de grano fino

3.3.3. Modelo de grano grueso o de islas

El cómputo paralelo se aplica de manera natural cuando se procesan islas (poblaciones), ya que cada uno de los individuos que pertenecen a cada una de ellas, es una unidad independiente.

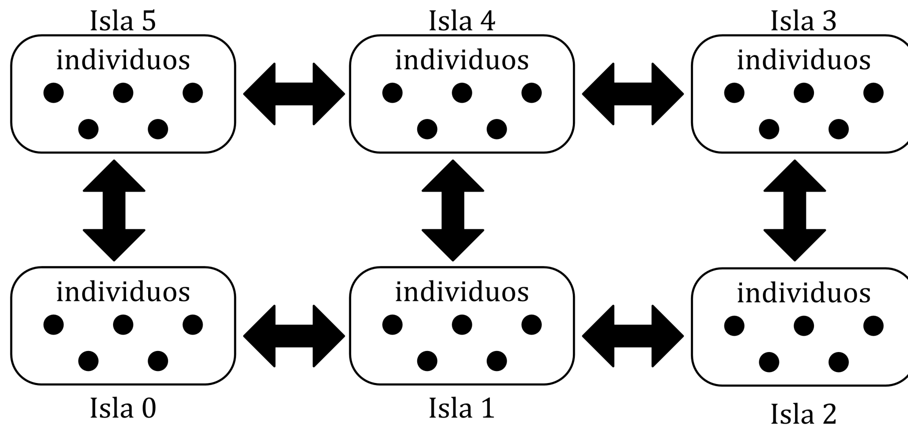


Fig. 3.7: Modelo de islas de Algoritmos Genéticos Paralelos

Debido a esto, el rendimiento de los algoritmos basados en islas se mejora con la inclusión del paralelismo [9, 10].

En la Fig. 3.7 se ilustra el modelo de islas de Algoritmos Genéticos Paralelos. Cada isla representa una población de individuos, la cual se hace evolucionar a través de los operadores de reproducción de Algoritmos Genéticos. Las islas, pueden compartir individuos entre sí con determinada frecuencia. Esto evita que el algoritmo caiga prematuramente en óptimos locales y favorece la diversidad de la población en búsqueda del óptimo global.

Al diseñar este modelo de AGP es necesario definir una serie de parámetros relativos a las migraciones los cuales se presentan a continuación.

- **Topología de intercambio:** Especifica para cada isla, a cual de ellas enviará individuos y de cual puede recibir.
- **Número de migrantes:** Indica el numero de individuos que enviará y recibirá cada una de las islas.
- **Política de selección de migrantes:** La política de selección indica de manera determinista o estocástica como se seleccionan individuos emigrantes desde la isla fuente.
- **Política de reemplazo:** Esta define como se integran los inmigrantes en la población.
- **Criterio de decisión de migración:** La migración puede ser decidida periódicamente o de acuerdo a un criterio.

La solución que resulta de estos modelos es diferente al resultado que produce un modelo de AG secuencial. Esto se debe principalmente a que las operaciones que se producen en cada subpoblación, la cual tiene menos individuos que la población global, son diferentes. Esto es, como

se trabaja con poblaciones más pequeñas, se logra converger más rápido que los secuenciales, y como consecuencia se reduce el tiempo de ejecución.

Como se trabaja con poblaciones pequeñas, se puede pensar en la falta de diversidad en las soluciones. Sin embargo, el análisis realizado en trabajos previos, ha demostrado que esto no ocurre, ya que el intercambio de individuos entre las subpoblaciones evita que esto suceda. Es así como estos algoritmos además de ser más rápidos, también alcanzan mejores soluciones que las ejecuciones secuenciales [35].

3.3.4. Modelos Híbridos o Jerárquicos

Al hacer la combinación entre los modelos anteriormente expuestos, surge el modelo híbrido, o jerárquico [9]. Este es llamado así debido a que combina los beneficios de los modelos por los que está compuesto y su propósito es mejorar el rendimiento que por si solo alguno de ellos no presenta.

En este sentido, el primer modelo híbrido es una combinación de un algoritmo de grano grueso en el nivel superior con un algoritmo de grano fino en el nivel inferior como se muestra en la Fig. 3.8. Existen varias implementaciones de este tipo de algoritmos. Por ejemplo, un Algoritmo Genético multipoblación cuyas poblaciones estén conectadas en forma toroidal de dos dimensiones y donde cada una de las poblaciones sea organizada siguiendo una estructura de grano fino, por ejemplo, una cuadrícula de dos dimensiones. Otra implementación es conectar las poblaciones del nivel superior por medio una topología de anillo, y conservar la distribución en cuadrícula de dos dimensiones para el nivel inferior.

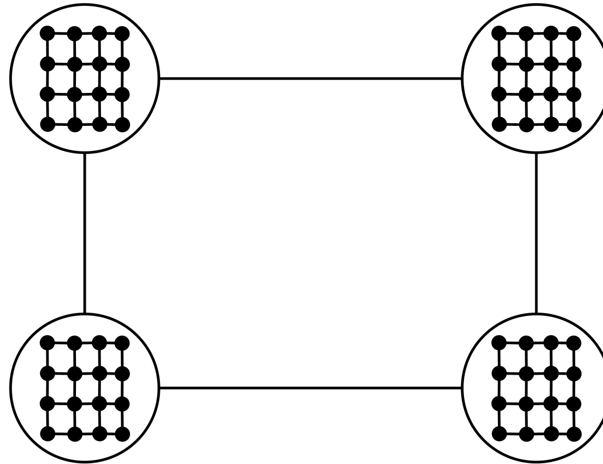


Fig. 3.8: Modelo híbrido con modelo de grano grueso en el nivel superior y de grano fino en el nivel inferior

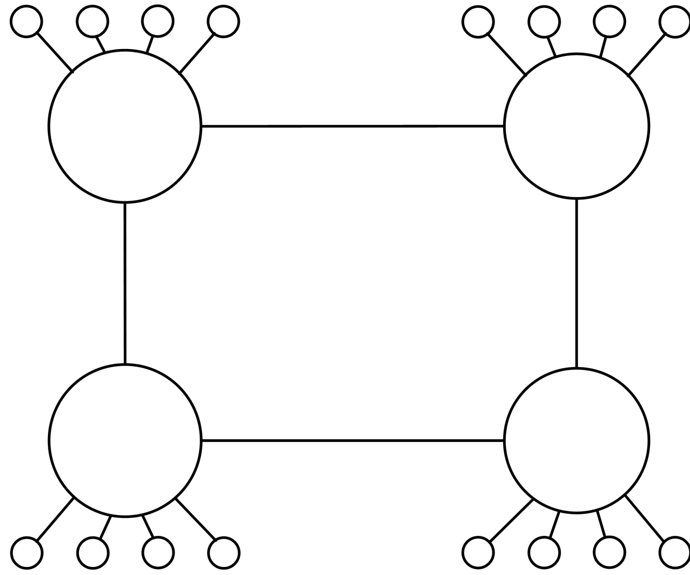


Fig. 3.9: Modelo híbrido con modelo de grano grueso en el nivel superior y maestro-esclavo en el nivel inferior

Otro tipo de jerarquía del modelo híbrido utiliza un modelo maestro esclavo en cada uno de las poblaciones del nivel superior. Esto se muestra en la Fig. 3.9. La migración se produce entre cada población, y la evaluación de los individuos se maneja en paralelo. Este enfoque no introduce nuevos problemas analíticos, y puede ser útil cuando se trabaja con aplicaciones complejas con funciones objetivo que necesitan una cantidad considerable de procesamiento [9].

Un tercer tipo de AGP híbridos utiliza un AGP de grano grueso en ambos niveles. Este modelo se muestra en la Fig. 3.10. En el nivel inferior se tiene una tasa alta de migración y tasa baja en el nivel alto.

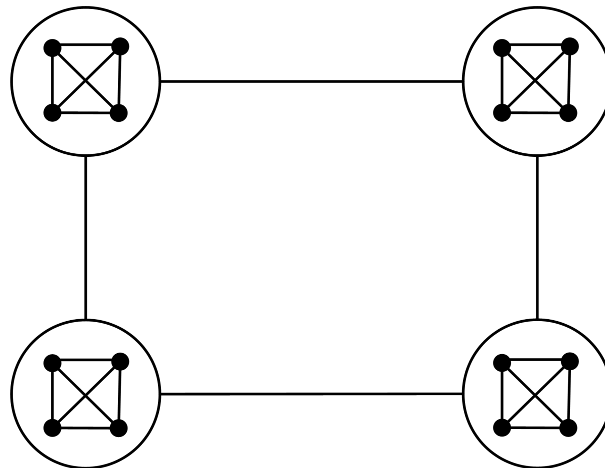


Fig. 3.10: Modelo híbrido con modelo de grano grueso en el nivel superior e inferior

Las implementaciones jerárquicas pueden reducir el tiempo de ejecución a diferencia de la implementación por si solo de uno de los modelos por los que está compuesto. Se ha encontrado que las combinaciones apropiadas de modelos de distintos tipos pueden mejorar los resultados del algoritmo, aunque es necesario ajustar adecuadamente el conjunto de parámetros que surgen en estos modelos [35].

3.4. Algoritmos Genéticos para el PAV

A continuación se describen las diferentes técnicas que pueden ser aplicadas a cada uno de los procesos que conforman el AG para resolver el Problema del Agente Viajero. Se expone como se puede implementar un AG a partir del modelo básico.

3.4.1. Codificación

Anteriormente, se mencionó que en la mayoría de los trabajos sobre Algoritmos Genéticos estudiados durante este trabajo de investigación, los individuos de la población son representados por cadenas binarias. Sin embargo, en el contexto del PAV esta codificación no es tan natural, por ejemplo, para el problema de cinco ciudades y 20 aristas que se ilustra en la Fig. 3.11, un individuo con codificación binaria representado en la Fig. 3.12 representa la ruta que utiliza las aristas 2, 9, 10, 12, y 15 para conectar las cinco ciudades como se muestra en la Fig. 3.13.

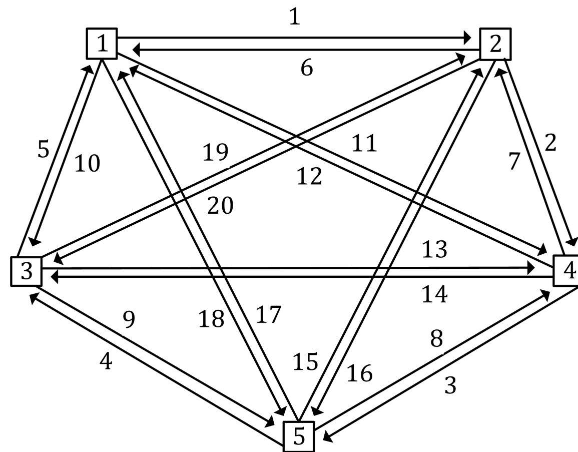
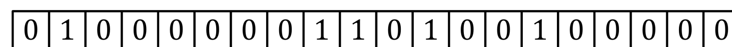


Fig. 3.11: Problema de 5 ciudades y 20 aristas.



0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	0	0

Fig. 3.12: Cromosoma con codificación binaria.

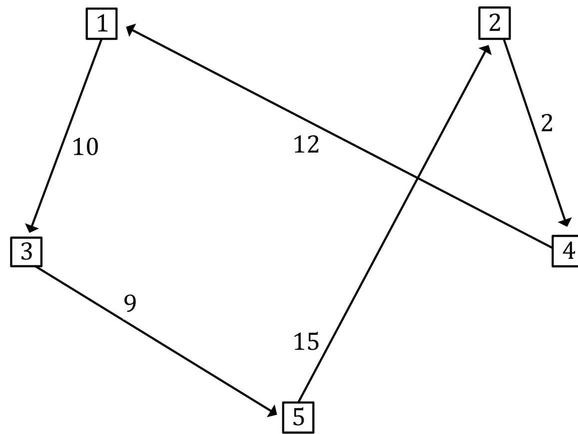


Fig. 3.13: Ruta que representa el cromosoma de la Fig. 3.12.

Una representación de este tipo no es la más adecuada para el PAV, ya que existe una gran cantidad de cromosomas con cinco "1" y quince "0" que no representan soluciones factibles para el PAV por no formar un circuito válido, por ejemplo: el cromosoma de la Fig. 3.14 representa el recorrido inválido de la Fig. 3.15.

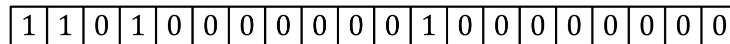


Fig. 3.14: Cromosoma de un recorrido inválido

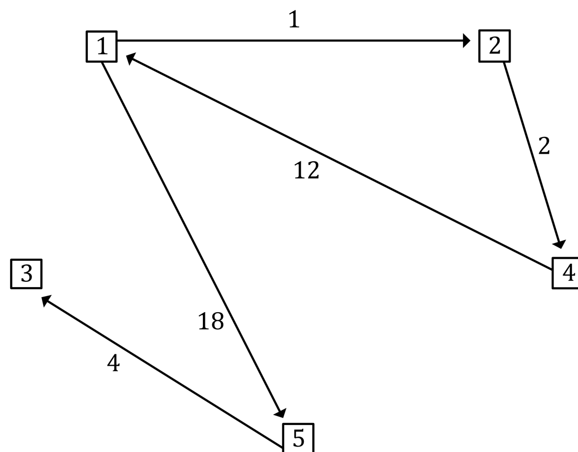


Fig. 3.15: Recorrido que representa el cromosoma inválido de la Fig. 3.14

Otra forma de representar los recorridos en forma binaria para un PAV de N ciudades es la siguiente:

- Cada ciudad se puede codificar como una cadena de $[\log_2 N]$ bits.

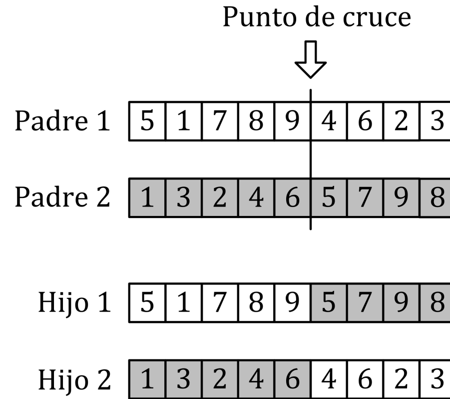


Fig. 3.16: Cruce externo clásico de un punto

- Un cromosoma es una cadena de $N[\log_2 N]$ bits.

Si se lleva a cabo una operación de mutación para este tipo de codificación, puede dar como resultado una secuencia de ciudades que no sea un recorrido válido. Por ejemplo, para un problema de veinte ciudades, son necesarios 5 bits para representar una ciudad:

- Puede obtenerse la misma ciudad dos veces en un cromosoma.
- Algunas series de 5 bits (por ejemplo, 10101) no corresponden a alguna ciudad.

Problemas similares se presentan cuando se aplica la operación de cruce. A consecuencia de esto, la mejor manera de codificar las soluciones del PAV, es una representación de ruta. Ésta consiste en una lista de números en orden correspondiente a las ciudades de un recorrido. Por ejemplo, para un problema con nueve ciudades el recorrido 5-1-7-8-9-4-6-2-3 representa el individuo 5, 1, 7, 8, 9, 4, 6, 2, 3. Nótese que el individuo 8, 9, 4, 6, 3, 5, 1, 7 también representa al mismo recorrido, sólo que inicia por la ciudad 8; lo mismo ocurre con el individuo 3, 2, 6, 4, 9, 8, 7, 1, 5, en el cual solamente se cambia el sentido del recorrido.

Utilizando este tipo de representación, el espacio de búsqueda estará formado por todas las permutaciones de las N ciudades a visitar y cada una de las permutaciones representa un recorrido candidato a ser la solución del problema.

El problema de esta codificación radica en que el cruce externo clásico entre genes, casi nunca proporciona individuos factibles debido a que se pueden repetir ciudades que ya están contempladas en la ruta. Por este motivo se debe recurrir a otros operadores de cruce y mutación que se utilizan específicamente para problemas de optimización combinatoria con permutaciones y que se verán más adelante.

En la Fig. 3.16 se demuestra como al hacer el cruce externo clásico se pueden obtener individuos que no cumplen con las restricciones del Problema del Agente Viajero.

3.4.2. Población inicial

La población inicial de un AG puede ser creada de diferentes maneras:

- Puede aceptar algunas salidas de un algoritmo voraz para el PAV que inicia en diferentes ciudades ó,
- la población puede inicializar con una muestra aleatoria de las permutaciones de $(1, 2, ..., N)$.

3.4.3. Función de aptitud

La evaluación de un cromosoma es directa: para cualquier solución potencial (una permutación de las ciudades), se calcula la distancia total del recorrido, iniciando y terminando en la misma ciudad.

3.4.4. Selección

La selección debe dirigir el proceso de búsqueda en favor de los individuos con mejor aptitud. Como se mencionó en el Capitulo 2, existen mecanismos de selección muy variados. El criterio para elegir un mecanismo de muestreo depende del problema y del buen juicio del programador. Para el PAV en el presente estudio se utilizó el método de selección por torneo.

3.4.5. Operadores de cruce

Hasta hace poco, tres operadores de cruce fueron definidos para la representación de ruta y estos mismos pueden ser aplicados al PAV [35]: cruce por emparejamiento parcial (PMX), cruce por orden (OX) y cruce por ciclos (CX).

Cruce por emparejamiento parcial (PMX)

El cruce por emparejamiento parcial consiste en elegir un tramo de la ruta de uno de los padres y cruzar manteniendo el orden y la posición de la mayor cantidad posible de ciudades del otro.

Dados dos padres, este operador de cruce copia una subcadena de uno de ellos directamente en las mismas posiciones del otro. Las posiciones restantes, se llenan con los valores que aún no hayan sido utilizados, en el mismo orden que se encontraban en su progenitor.

Por ejemplo, considerar los individuos padres que se muestran en la Fig. 3.17(a). Para generar los descendientes, se eligen aleatoriamente las subcadenas comprendidas entre dos puntos de corte que definen los tramos y se intercambian. Esto puede verse en la Fig. 3.17(b). En cada una de las posiciones faltantes de los hijos se colocan los valores de los padres que no están ya

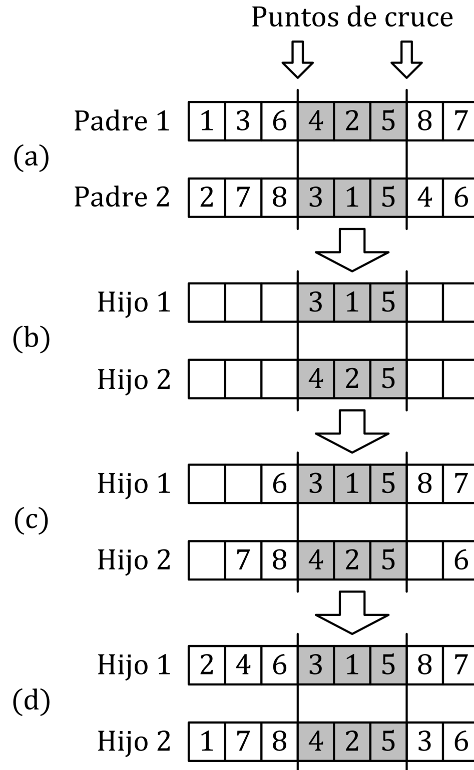


Fig. 3.17: Cruce por emparejamiento parcial (PMX)

en el hijo, como se observa en la Fig.3.17(c). Los valores que faltan se resuelven mediante los valores emparejados en las subcadenas iniciales, y se obtienen los dos individuos que se ilustran en la Fig.3.17(d).

Cruce por orden (OX)

El cruce por orden consiste en copiar en cada uno de los hijos una subcadena de uno de los padres mientras se mantiene el orden relativo de las ciudades que aparecen en el otro padre.

Se tienen los dos individuos padres que se observan en la Fig. 3.18(a). Para generar los descendientes se eligen las subcadenas comprendidas entre dos puntos de corte aleatorios, mismos que se muestran en la Fig. 3.18(b).

A continuación, se copian los valores que hacen falta del padre contrario. Se inicia a partir de la zona copiada y se respeta el orden sin repetir elementos. Es así como se obtienen finalmente los dos individuos mostrados en la Fig. 3.18(c).

Cruce por ciclos (CX)

El cruce por ciclos, consiste en generar los hijos de tal manera que cada ciudad y su posición se hereden sucesivamente de alguno de los padres, de acuerdo con las posiciones que tienen dentro de un ciclo.

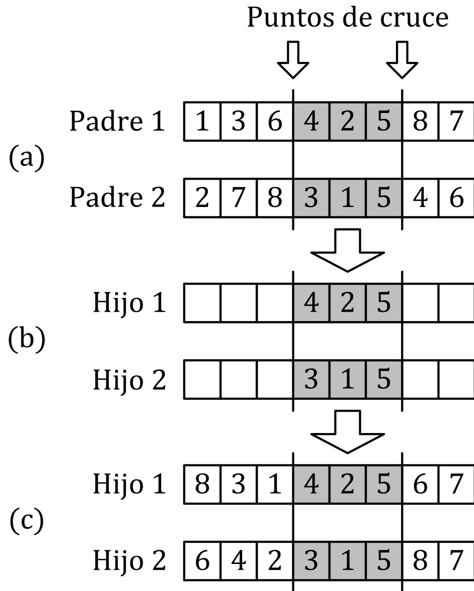


Fig. 3.18: Cruce por orden (OX)

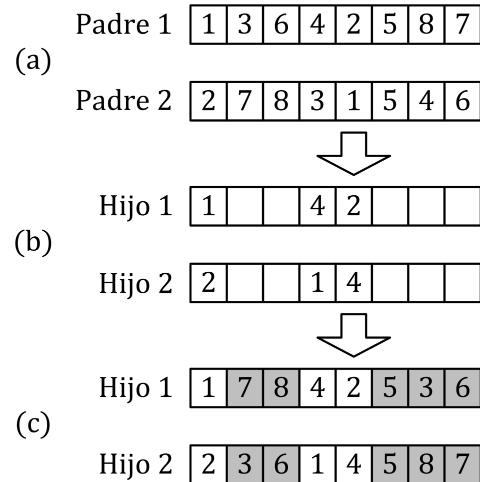


Fig. 3.19: Cruce por ciclos (CX)

Por ejemplo, si se toma en cuenta a los dos individuos padres mostrados en la Fig. 3.19(a), se define un ciclo por el primer elemento de Padre 1 y alternando con el homólogo en Padre 2 hasta completar dicho ciclo. En este caso, el ciclo obtenido es el siguiente: en Padre 1 se selecciona la ciudad 1 que corresponde con la ciudad 2 en Padre 2. La ciudad 2 en Padre 1 se corresponde con la ciudad 4 en Padre 2 y la ciudad 4 en Padre 2 se corresponde con la ciudad 1. Esto cierra un ciclo 1→2→4.

Para generar el Hijo 1 se copian los valores de Padre 1 que son parte del ciclo (Fig. 3.19(b)), y para completar Hijo 1, se obtienen las ciudades restantes del otro padre. El Hijo 2 se obtiene mediante el mismo procedimiento. Los hijos resultantes se pueden ver en la Fig. 3.19(c).

3.4.6. Operador de mutación

Al igual que en la etapa de cruce para el PAV, en la etapa de mutación también resultan individuos no válidos en caso de que se aplicara el operador de mutación clásico, debido a que se tendría que modificar algún gen del cromosoma. El valor del nuevo Gen se repetiría en el cromosoma, lo cual no es permitido por las restricciones del PAV. A continuación, se muestran los tipos de mutación que se pueden emplear para el Problema del Agente Viajero.

Mutación por inversión

Es interesante hacer notar que el operador de inversión sí produce individuos factibles bajo el tipo de representación de ruta. La inversión simple selecciona al azar dos puntos a lo largo de un individuo e invierte el orden de los genes intermedios. Este tipo de operador de mutación

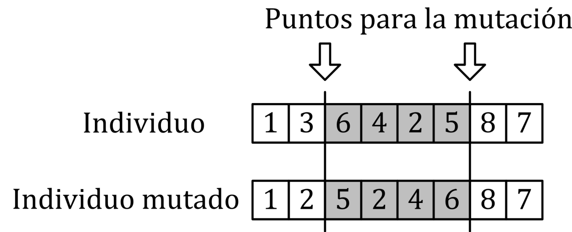


Fig. 3.20: Mutación por inversión de un individuo

se muestra en la Fig. 3.20.

La mutación por inversión es un operador unitario, este operador por sí solo no tiene el mismo desempeño que los operadores de cruce (CX, OX, PMX), por lo que debe complementarse con alguno de ellos.

Mutación por intercambio

En este tipo de mutación se seleccionan dos posiciones al azar del individuo a mutar y se intercambian los valores de dichas posiciones. Por ejemplo, en la Fig. 3.21 puede observarse como actúa este operador.

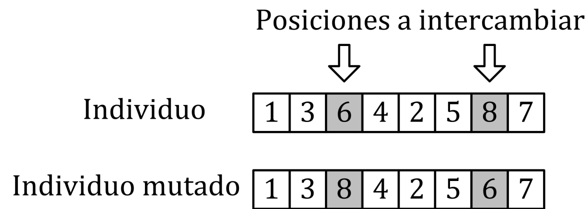


Fig. 3.21: Mutación por intercambio de un individuo

Mutación por inserción

En este tipo de mutación se inserta una o varias ciudades elegidas al azar en ciertas posiciones también elegidas al azar. Puede haber una o varias inserciones tal y como se muestra en la Fig. 3.22.

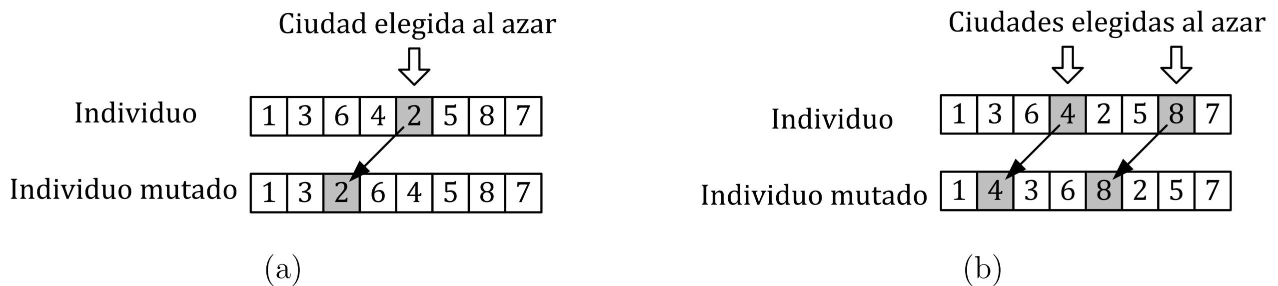


Fig. 3.22: Mutación por inserción. (a) Una inserción, (b) Dos inserciones.

Con una sola inserción, por ejemplo, se selecciona la ciudad 2 y se inserta en la tercera posición, desplazando todos los elementos hacia la derecha.

Capítulo 4

Implementación de Algoritmos Genéticos Paralelos

4.1. Introducción

El objetivo del paralelismo es ejecutar un programa en menos tiempo utilizando varios procesadores. La idea central es dividir un problema grande en varios mas pequeños y repartirlos entre los procesadores disponibles. A continuación se describe como se resuelve el Problema del Agente Viajero con Algoritmos Genéticos Paralelos en un procesador multinúcleo y en una GPU.

Un individuo en términos de Algoritmos Genéticos representa una ruta para el PAV, que a su vez, es una posible solución de dicho problema. Como se explicó en la sección 3.4.1 la mejor codificación de las soluciones para el PAV, es una representación de ruta en la cual a cada ciudad que comprende un recorrido se le asigna un número entero positivo. Por ejemplo, para un problema con 9 ciudades el recorrido 5-1-7-8-9-4-6-2-3-5, representa una ruta que inicia en la ciudad número 5, luego visita la número 1 y así sucesivamente para terminar en la ciudad 3 y regresar a la ciudad de inicio. Ésta representación es la que se utiliza en este trabajo de investigación.

Para generar la población inicial, se puede hacer de manera aleatoria o utilizar un algoritmo voraz que genere rutas mas cercanas a la óptima. Si las rutas de la población inicial se generan de manera aleatoria, la mayoría de estas resultan muy alejadas del óptimo, y el AG tarda más tiempo en encontrar mejores soluciones. En cambio, si los individuos de la población inicial se generan con un algoritmo voraz, se obtienen mejores rutas más rápido y por ende una población inicial que contiene rutas más cercanas a la óptima. Con esto, se acelera la convergencia del Algoritmo Genético.

En este trabajo se generó la población inicial con el algoritmo Greedy [39, 40, 41]. El algorit-

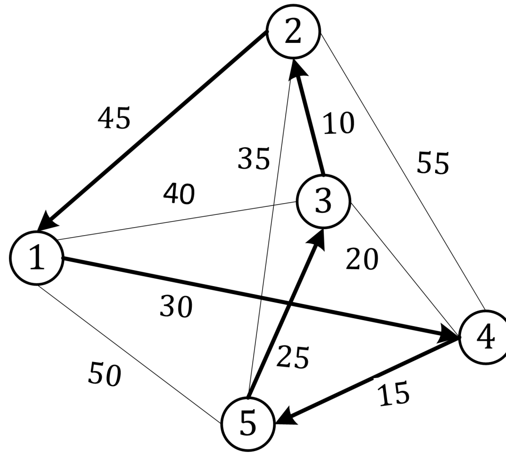


Fig. 4.1: Funcionamiento del algoritmo Greedy.

mo consiste en siempre tomar el óptimo local y proceder desde ahí. La acumulación secuencial de estos óptimos locales nos arrojan la respuesta.

En la Fig. 4.1 se muestra el funcionamiento del algoritmo Greedy para el Problema del Agente Viajero con 5 ciudades. Para este caso la ruta obtenida inicia en la ciudad 1 y es 1-4-5-3-2-1.

Al momento de resolver el problema, el algoritmo Greedy siempre elige la ciudad que está mas próxima en cada momento y se descartan las ciudades que ya fueron elegidas [40]. Posteriormente se regresa a la ciudad donde se inició. Con esto se cierra la ruta cumpliendo así con la restricción del problema del agente viajero de pasar una sola vez por cada ciudad y volver a la ciudad de inicio.

Después de crearse la población inicial de posibles soluciones, se asigna la aptitud (que tan buena solución es cada una de ellas) a cada uno de los individuos. Para utilizar un AG como un procedimiento de búsqueda es necesario definir una función de aptitud que evalúe correctamente las reglas de decisión generadas por el algoritmo de aplicación.

Para este problema la función de aptitud es la distancia que se recorre siguiendo las rutas que representan los individuos generados. Mientras menor sea la distancia, mejor es el individuo. De esta manera, si la distancia del recorrido que representa un individuo 1 es menor que la de un individuo 2, se dice que el individuo 1 es mejor que el 2.

Con la función de aptitud, se calcula la distancia recorrida que representa cada uno de los individuos. De acuerdo al valor que cada uno tenga, se toman las decisiones adecuadas que permiten evolucionar a la población en búsqueda de mejores individuos. Mientras menor sea la distancia recorrida por cada individuo, más cerca se está de la ruta óptima.

Para obtener los individuos que deben cruzarse se utilizó el tipo de selección por torneo. Con esta técnica se seleccionan de forma aleatoria un número determinado de individuos de la población. De estos, se identifica a los dos mejores según su aptitud. Los dos individuos

seleccionados sirven como padres en la etapa de cruce.

El operador de cruce combina propiedades de dos individuos de la población anterior para crear nuevos individuos [35]. El tipo de cruce que se utiliza para las implementaciones paralelas del algoritmo es el cruce por orden. La cantidad de individuos a cruzar dentro del AG es manejado mediante el porcentaje de cruce (P_c). Se ejecutan múltiples operaciones de cruza simultáneamente. En caso de que los nuevos individuos sean mejor que los individuos padres, éstos pasarán a ser parte de la población (los nuevos individuos reemplazan a los padres).

El operador de mutación crea un nuevo individuo realizando algún tipo de alteración usualmente pequeña, en un individuo de la población anterior [35]. Esto sirve para evitar la pérdida de diversidad producto de genes que han convergido en un cierto valor para toda la población y por lo tanto no pueden ser recuperadas por el operador de cruce.

El operador de mutación se emplea con una probabilidad P_m que indica si la mutación se hace o no. Para los algoritmos implementados en paralelo se utiliza la mutación por inversión. En cada iteración, cada hilo de ejecución muta un individuo y se obtiene uno nuevo. En caso de que la distancia recorrida por la ruta que representa el nuevo individuo sea menor que la del individuo a mutar, el nuevo individuo pasa a ser parte de la población y el individuo que sirvió para hacer la mutación, es eliminado.

El Algoritmo Genético implementado se detiene cuando termina de ejecutar un número determinado de iteraciones. Este número se define como parámetro del algoritmo al inicio del mismo.

4.2. Implementación de un Algoritmo Genético Paralelo en un procesador multinúcleo

Varias plataformas de computo paralelo, en particular las plataformas multinúcleo, ofrecen un espacio de direcciones compartido. Un modelo de programación para estas arquitecturas es un modelo de hilos, en el cual todas los hilos tienen acceso a variables compartidas que se utilizan para intercambiar información y datos [2].

La programación por hilos permite crear varios hilos de ejecución para nuestro sistema, tal que cada hilo ejecuta una tarea distinta en cada procesador. La ventaja principal de este tipo de programación es el uso de memoria compartida, por lo que los hilos de un proceso comparten variables globales y pueden cooperar entre ellos. Sin embargo, debe tomarse en cuenta que la creación y destrucción de hilos a lo largo del proceso tarda cierto tiempo que se suma al tiempo que tarda el algoritmo. La creación de hilos nos permite abarcar un mayor número de soluciones con lo cual tenemos más probabilidad de encontrar una solución mas cercana a la óptima.

Para utilizar hilos en C, podemos utilizar la librería pthreads que implementa el estándar

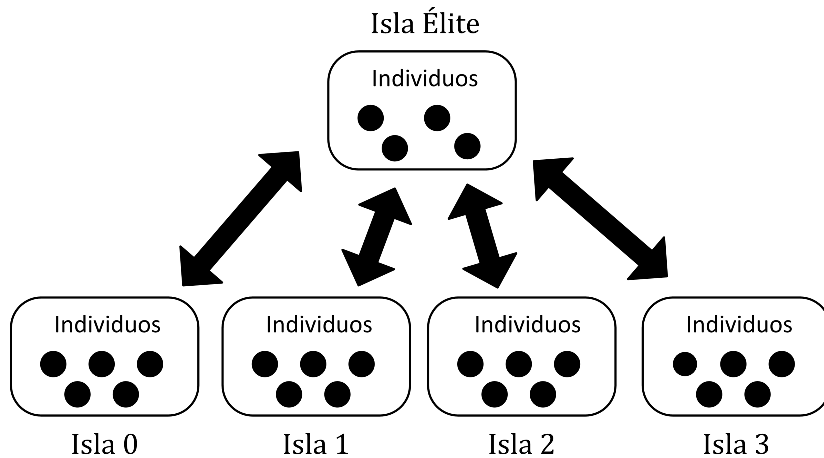


Fig. 4.2: AGP con isla Élite

POSIX (Portable Operating System Interface). Con ésta, se podrán crear los hilos necesarios para poder particionar el código en piezas y que se ejecute en paralelo en hilos diferentes [42].

La granularidad se refiere al tamaño de las piezas en que se divide una aplicación. Dichas piezas pueden ser una sentencia de código, una función o un proceso que se ejecutarán en paralelo. De acuerdo a la granularidad, el paralelismo se puede clasificar en grano fino y grano grueso.

En el paralelismo de grano fino, el código se divide en una gran cantidad de piezas pequeñas. Un ciclo se divide en varios sub-ciclos que se ejecutarán en paralelo (paralelismo de datos).

El paralelismo de grano grueso es a nivel de subrutinas o segmentos de código, donde las piezas son pocas. Se le conoce como paralelismo de tareas.

Debido al paralelismo de los AG, es posible incrementar el tamaño de la población, reducir el costo computacional y mejorar el desempeño de los AG.

En el AGP por modelo de islas descrito anteriormente en la sección 3.3.3 se le llama isla a cada una de las subpoblaciones que se forman al dividir la población total. Cada isla ejecuta su propio algoritmo genético independientemente de las demás. La única comunicación que tienen las islas entre sí, se dá cuando después de determinadas generaciones, cada una de las islas, envía a sus mejores individuos a otra isla así como también recibe individuos de alguna otra (según la topología dada). Cada isla eliminará a sus peores individuos y los reemplaza por los que recibió desde otra. Esto se hace con el fin de promover la diversidad de individuos en cada isla y evitar estancarse en óptimo locales. A esta etapa de intercambiar individuos entre islas se le conoce como migración en términos de Algoritmos Genéticos Paralelos.

Llamaremos Algoritmo Genético Paralelo con isla Élite al algoritmo propuesto en este trabajo de investigación, el cual está basado en el modelo de islas de AGP y se muestra en la Fig. 4.2. Al igual que en el modelo de islas tradicional, cada subpoblación es una isla dentro de las

Algoritmo 2 Pseudocódigo del AGP en hilos POSIX

```
generar población inicial
evaluar a los individuos de la población
migrar individuos a la isla Élite
while TERMINO = FALSO do
   seleccionar individuos
   cruzar individuos seleccionados
   mutar individuos de la población
   evaluar la adaptación de los descendientes obtenidos por cruce y mutación
   reemplazar individuos en caso de que los nuevos sean mejores
   migrar individuos a la isla Élite
end while
identificar mejor solución presente en la isla Élite
return reportar mejor solución
```

cuales se tiene cierta cantidad de individuos que a su vez son parte de la población total. La diferencia entre el algoritmo tradicional y el algoritmo propuesto, es que las islas no comparten individuos entre sí, es decir, la migración no existe. Sin embargo, se define una isla Élite que solo contendrá al mejor individuo de cada una de las islas.

Las islas en cada iteración, identifican a su mejor individuo y lo envían a la isla Élite. El algoritmo está diseñado de tal manera que existe la posibilidad que cualquier isla pueda cruzar alguno de sus individuos con algún individuo de la isla Élite. Esto agrega diversidad a la búsqueda debido a que se hace posible que el cruce puede existir entre individuos de distintas islas. La comunicación entre las islas, solo son posibles a través de la isla Élite.

Se crearon un número determinado de hilos de ejecución y se dividió la población total en un número determinado de islas de tal manera que cada hilo trabaja sobre una isla. El Algoritmo 2 es ejecutado por cada hilo creado. Al inicio, cada hilo crea a los individuos de la isla, mismos que posteriormente mejora a través de cruce y mutación.

Cuando dentro de una isla es encontrado un mejor individuo como resultado del operador de cruce o el operador de mutación, éste se envía a la isla Élite desde donde las otras islas pueden utilizarlo para cruzarlo con alguno de sus individuos. De esta manera es como individuos pertenecientes a una isla, pueden ser mejorados a través de individuos ajenos a la misma.

El algoritmo termina cuando todos lo hilos han ejecutado el número de iteraciones que se definió, y la mejor solución final es el mejor individuo que se encuentre en la isla Élite.

4.3. Algoritmos Genéticos Paralelos en GPU

Anteriormente, a lo largo de este proyecto de investigación, se ha hablado de la paralelización de AG [10]. Sin embargo, hasta el momento en el Algoritmo Genético Paralelo que se

implementó anteriormente en hilos POSIX, se utilizó paralelismo al momento de hacer evolucionar un algoritmo genético en cada hilo POSIX que se ejecuta en la CPU. De esta manera, el paralelismo se muestra sólo al momento de ejecutar de manera simultanea cada uno de los hilos, ya que todas las operaciones del Algoritmo Genético se hacen de manera secuencial en cada uno de los hilos POSIX.

Ahora bien, los pasos del Algoritmo Genéticos, incluso los operadores genéticos, pueden ser paralelizados en la implementación. Por ejemplo, Fujimoto y Tsutsui paralelizaron el operador de cruce OX en [12].

La creación de la población inicial también puede ser paralelizada. Se pueden crear de manera simultanea a cada uno de los individuos que formarán parte de la población inicial, y evitar así crearlos mediante un ciclo. De la misma manera, la etapa de evaluación, y los operadores cruce y mutación, pueden hacerse para cada individuo o para cada grupo de individuos respectivamente de forma independiente.

En el caso de la implementación de algoritmos genéticos paralelos donde se incluye el operador genético de migración (como el caso de modelo de islas), esta operación también puede ser paralelizada, ya que no existe dependencia de datos entre los individuos que se emigran de una subpoblación a otra. La implementación de la paralelización de los operadores genéticos antes mencionados, reduce el tiempo de ejecución del algoritmo.

4.3.1. Unidades de Procesamiento Gráfico

El desarrollo de aplicaciones en paralelo ya no está restringido solo a aplicaciones que se realizan en grandes y costosos equipos de cómputo. En los años recientes prácticamente todos los modelos de computadoras personales cuentan con al menos un procesador con dos núcleos y, en algunos casos, poseen tarjetas gráficas con muy alta capacidad computacional que pueden llevar a cabo operaciones de propósito general.

Una unidad de procesamiento gráfico (GPU, del inglés Graphics Processing Unit) es un procesador multinúcleo que tiene la capacidad de ejecutar una gran cantidad de instrucciones de manera concurrente, a través del uso de un gran número de hilos de ejecución. El GPU funciona como coprocesador en una computadora cuyo procesador central es un CPU [3, 4].

Al ser el GPU un procesador multinúcleo, tiene como objetivo mejorar la velocidad de ejecución de los programas secuenciales y al mismo tiempo incorporar los beneficios del cómputo en paralelo [43].

4.3.2. Arquitectura CUDA

La arquitectura CUDA (Compute Unified Device Architecture) surgió en 2006, cuando NVIDIA lanzó al mercado sus tarjetas GeForce 8800 GTX. Éstas, fueron las primeras que incluyeron

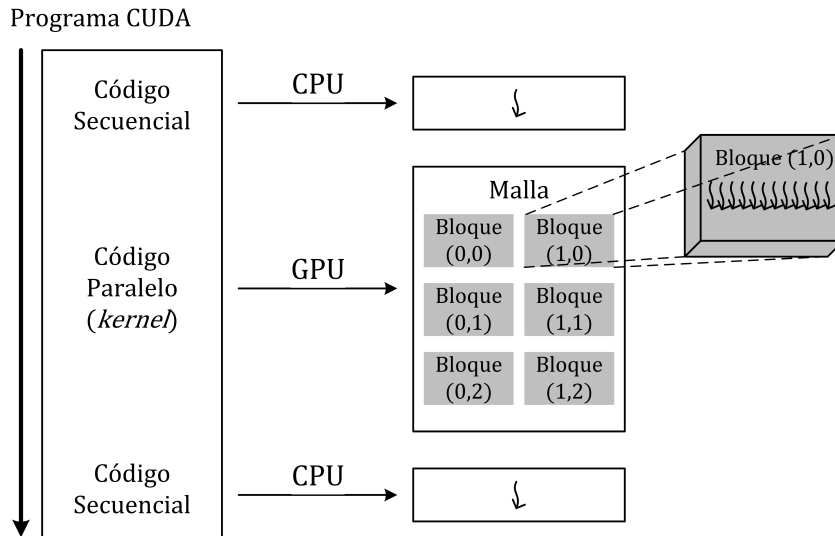


Fig. 4.3: Modelo de programación CUDA

la capacidad de utilizarse para la solución de problemas de propósito general. CUDA intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten la ejecución simultánea de miles de hilos.

CUDA es una extensión del lenguaje de programación C/C++ que involucra un modelo de programación y un conjunto de instrucciones que se utilizan para desarrollar aplicaciones en GPUs [4]. Este conjunto de instrucciones permiten que los programadores desarrollen aplicaciones que comunican a la CPU con la GPU para utilizar de esta última sus multiprocesadores, sobre los cuales pueden ser ejecutadas simultáneamente varias tareas. Además del lenguaje C, CUDA soporta otros lenguajes de programación como Fortran, Java o Python. En este trabajo de investigación se utiliza CUDA con lenguaje de programación C para desarrollar los algoritmos de prueba.

El modelo de programación de CUDA se muestra en la Fig. 4.3. Éste, aprovecha el paralelismo así como un mayor ancho de banda de la memoria en las GPU. Además, es un modelo escalable que permite la implementación de aplicaciones en dispositivos con diferente número de núcleos [4]. Para ello, maneja tres conceptos básicos: jerarquía de grupos de hilos, memoria compartida y barreras de sincronización. Basado en estos conceptos, el modelo permite al programador dividir un problema en problemas más pequeños a los que se les llama tareas, donde cada una de ellas será asignada y ejecutada en bloques de hilos independiente. Éstos, a su vez se ejecutan en los núcleos disponibles del GPU, en cualquier orden y de forma concurrente o secuencial.

En este contexto, se llama *kernel* a una subrutina que se ejecuta N veces en paralelo en N número de hilos en la GPU. El *kernel* se ejecuta basado en el modelo programa único y datos múltiples (SPMD por sus siglas en inglés para *Single-Program Multiple-Data*), por lo que es

necesario definir de manera manual el número de hilos que se desea ejecuten el mismo programa [43, 44].

Un programa en CUDA tiene secciones de código secuencial para el CPU y código paralelo para el GPU. Al inicio, el hilo principal se ejecuta secuencialmente en el CPU. Cuando se ejecuta un *kernel*, el CPU pasa el mando del programa al GPU donde un gran número de hilos son ejecutados concurrentemente para realizar operaciones en paralelo. Al conjunto de hilos generados en un *kernel* se le denomina malla (en inglés es llamado *grid*). Ésta a su vez está formada por bloques de hilos donde cada bloque tiene la capacidad de procesar 1024 hilos como máximo para la arquitectura actual de GPU. Cuando todos los hilos del *kernel* terminan de ejecutarse, se pasa de nuevo el mando al CPU y el hilo principal continua con su ejecución secuencial en el CPU [3, 4].

Un *kernel* es ejecutado por múltiples bloques. Éstos, son de la misma dimensión comparados entre sí y por lo cual contienen la misma cantidad de hilos. Entonces, el número total de hilos que ejecutan un kernel es igual al número de bloques multiplicado por el número de hilos que contiene cada bloque.

Cada hilo está identificado con un identificador único que se accede con la variable threadIdx. Así mismo, cada bloque puede ser identificado mediante blockIdx. Los hilos dentro de un bloque pueden ser definidos en una, dos o tres dimensiones a criterio del programador. Cualquiera que sea la manera de distribuir los hilos en un bloque, como se dijo anteriormente, la cantidad de hilos en un bloque no debe ser mayor a 1024. Ésta limitación es una consecuencia de que todos los hilos de un bloque residen en el mismo procesador y deben compartir los recursos de memoria.

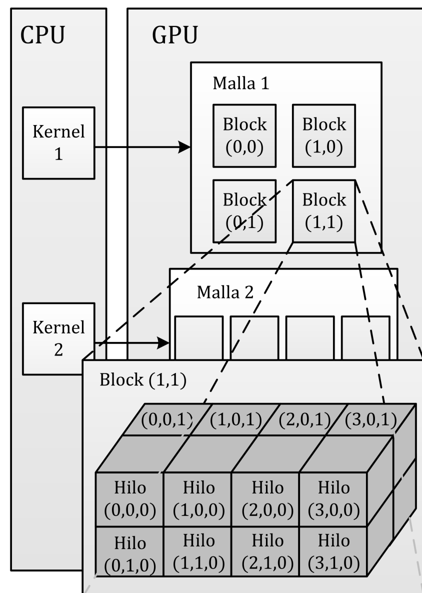


Fig. 4.4: Organización de los hilos en CUDA

En el caso de los bloques, cada uno de ellos puede ser identificado mediante blockIdx. A diferencia de los hilos, solo pueden ser definidos en una o dos dimensiones dentro de una malla de bloques. ThreadIdx, blockIdx así como la variable blockDim que contiene el tamaño del bloque (número de hilos que contiene un bloque) pueden ser accedidas y utilizadas por el programador si así lo requiere.

En la Fig. 4.4 se muestra como están organizados los hilos de una GPU con arquitectura CUDA. Cada bloque se ejecuta por los núcleos de un multiprocesador (SM, del inglés Streaming Multiprocessor). Cada SM está compuesto de varios procesadores SP (del inglés Streaming Processors) que son a los que llamamos núcleos de la GPU. El hecho de que cada multiprocesador ejecute un bloque de hilos, permite la escalabilidad en la programación, ya que no es necesario modificar un programa para que pueda ejecutarse en procesadores gráficos con distinta cantidad de multiprocesadores.

Al iniciar un programa en la GPU, los bloques de hilos son distribuidos a todos los multiprocesadores SM que se tengan disponibles. El multiprocesador, divide los hilos de un bloque en *warps*. Un *warp* equivale a 32 hilos que es la cantidad que un SM puede ejecutar a la vez [3, 4].

4.3.3. Memorias del GPU y transferencia de datos

En CUDA, el CPU y el GPU tienen espacios de memoria diferentes. Cuando se quiere ejecutar un *kernel*, el programador debe asignar memoria en el GPU y transferir los datos que requiera desde la memoria del CPU a la memoria del GPU. Del mismo modo, después de la ejecución del *kernel*, es necesario transferir los datos de los resultados desde la memoria del GPU a la memoria del CPU y liberar la memoria del GPU que ya no se necesita. El sistema de ejecución CUDA proporciona funciones de interfaz de programación de aplicaciones (API, del inglés *Application Programming Interface*) para llevar a cabo estas actividades en nombre del programador.

En la Fig. 4.5 se muestra una visión general del modelo de memoria del GPU según la arquitectura CUDA, donde se muestran la memoria global, memoria constante, memoria de textura, memoria compartida, memoria local y de registro del hilo. Es decisión de los programadores elegir la asignación, movimiento y uso que le vaya a dar a los distintos tipos de memoria. En la parte inferior de la figura, se observa la memoria global y la memoria constante. Estas son las memorias a las que el código en CPU puede transferir datos y desde las cuales el GPU puede hacer la acción inversa. Es por eso que se muestra como una comunicación bidireccional. Cabe señalar que la memoria constante permite accesos de solo lectura por el código en GPU, tal como lo indica la Fig. 4.5.

A continuación se describen los tipos de memoria del GPU que se utilizan con mayor fre-

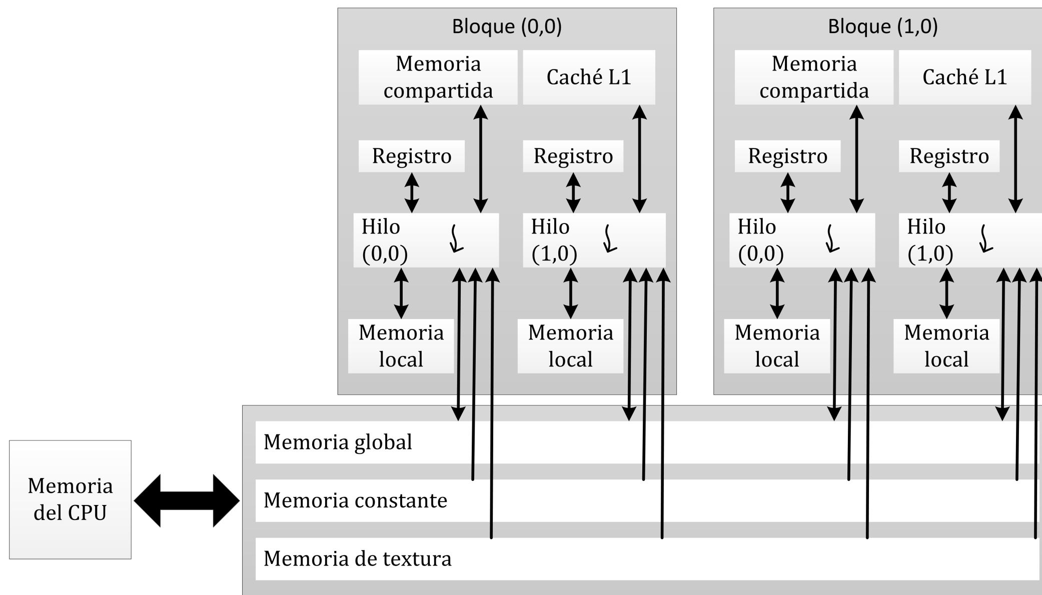


Fig. 4.5: Espacios de memoria en un GPU con arquitectura CUDA

cuencia en un programa en CUDA. Dentro de estas se encuentran los tipos que fueron utilizados en el algoritmo desarrollado del presente trabajo de investigación.

Memoria Global. La memoria global es la memoria principal y de mayor tamaño en el GPU. Esta es una memoria DRAM (del inglés, Dynamic Random Access Memory), de propósito general. Los datos que se almacenan en la memoria global se asignan y se liberan por el CPU y pueden ser accedidos por él mismo, así como por cualquier hilo del GPU. Sin embargo, los accesos que hace el CPU, solo pueden ser realizados a través de copias de memoria [3, 4, 43].

A diferencia de otros tipos de memoria presentes en el GPU, la memoria global se encuentra fuera del circuito integrado del GPU. Además, presenta una latencia alta, alrededor de cientos de ciclos de reloj son necesarios para accesar a ella. Debido a esta característica, esta memoria no es lo suficientemente rápida como para satisfacer la demanda de los multiprocesadores SM.

Además, la transferencia de datos desde la memoria de CPU a la memoria del GPU son relativamente lentas lo que produce más consumo de ancho de banda. Para contrarrestar estos problemas, como se menciona en [43], los datos dentro de los multiprocesadores pueden ser reutilizados. Para ello CUDA proporciona espacios de memoria configurables adicionales a la memoria global. Utilizar o no estos espacios de memoria configurables representa una diferencia importante en el rendimiento de los programas CUDA [44].

Memoria Local. Cada hilo del GPU tiene una memoria local que solo puede ser accedida por el propio hilo. Se utiliza para almacenar variables y cálculos privados de los hilos, y tiene un tamaño de hasta 512 Kilobytes. Este espacio de memoria es gestionada por el compilador y se asigna cuando no hay suficiente espacio en los registros para las variables locales que utilizan

los hilos o en caso de que un *kernel* utilice más registros de los disponibles en el SM [4, 44].

Memoria Compartida. La memoria compartida de un GPU está localizada dentro de cada multiprocesador SM y tiene un tamaño de hasta 48 Kilobytes [45]. Esta memoria, solo es accesible para los hilos que componen el mismo bloque dentro de un SM. Es mucho más rápida que la memoria global y la memoria local de cada hilo. Se utiliza para reducir el número de accesos de un hilo a la memoria global así como también puede ser muy útil para compartir datos entre los hilos de un bloque.

Cada hilo del bloque, debe cargar datos desde la memoria global a la memoria compartida. A partir de ese momento, cualquier hilo del bloque puede utilizar los datos presentes en la memoria compartida del bloque al que pertenece. Con esto, se reduce el número de accesos a la memoria global por parte de los hilos. Algunas veces, es necesario sincronizar todos los hilos dentro del bloque para evitar errores de actualización de datos en la memoria compartida. Al finalizar la ejecución, si los datos presentes en la memoria compartida necesitan ser mostrados o manipulados en la CPU, es necesario copiarlos primero desde la memoria compartida a la memoria global y luego desde la memoria global del GPU a la memoria de CPU [4, 44].

Cualquier oportunidad de sustituir los accesos a la memoria global por accesos a la memoria compartida debe ser aprovechada. Debido a que esta última es más rápida que la memoria global, nos puede ayudar a reducir el tiempo de ejecución de un programa en CUDA. Sin embargo, esto no siempre es posible, ya que para instancias de problemas grandes, posiblemente el tamaño de la memoria compartida no sea suficiente.

4.4. Paralelización del modelo de islas de AGP en GPU

Ejecutar el ciclo reproductivo de un AG simple sobre individuos con muchos genes (rutas con muchas ciudades en el caso del problema del agente viajero) y/o grandes poblaciones requieren muchos recursos computacionales. El cómputo paralelo se aplica de manera natural cuando se procesan poblaciones, ya que cada uno de los individuos que pertenecen a ella, es una unidad independiente. Debido a esto, el rendimiento de los algoritmos basados en poblaciones se mejora con la inclusion del paralelismo.

Por lo general con el fin de garantizar la diversidad de los individuos, los AG mantienen un grupo o población que consiste en un buen numero de individuos. Para tener una buena solución al problema, pueden crearse mas direcciones de búsqueda en un area mas grande. Esto es, aplicar el AG para cada grupo de individuos o subpoblaciones (islas). Así, cada AG está evolucionando a su grupo de individuos y buscando una solución óptima en el mismo.

La forma mas razonable de paralelizar el modelo de islas en un GPU con CUDA es el mapeo de actividades de los individuos para hilos separados. Ya que para todos los individuos se hará el mismo trabajo, cada uno de los hilos puede manipular las operaciones sobre cada uno de los

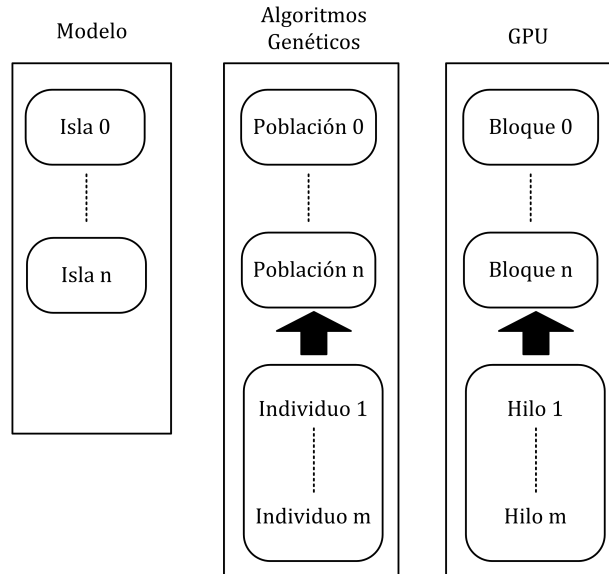


Fig. 4.6: Analogía entre algoritmos genéticos por modelo de islas y GPU

individuos.

La Fig. 4.6 muestra la analogía que se hace entre los conceptos del modelo de islas, algoritmos genéticos y la GPU. Esto nos ayuda a visualizar de que manera se relacionan estos tres conceptos dentro del algoritmo implementado.

El modelo de islas con isla Élite en hilos POSIX que se describió anteriormente, se implementó en GPU. Existen algunas diferencias en cuanto a la manera de como se ejecutan las instrucciones en cada uno de los modelos ya que la arquitectura del GPU nos permite ejecutar una mayor cantidad de hilos concurrentes en comparación con la programación en hilos POSIX.

En la Fig. 4.7 se muestra como se ejecuta una isla en particular en GPU. En este esquema, se representa una isla con un bloque de hilos. Un individuo es representado por un hilo y cada operador genético (selección, cruce o mutación) es separado por barreras de bloque para asegurar la sincronización entre los hilos [22].

Al tener el algoritmo completo en la GPU, no es conveniente estar comunicando constantemente el CPU con el GPU. Esta limitación se debe a que los multiprocesadores en GPU son usados de acuerdo al modelo SPMD, entonces, la misma configuración de parámetros y los mismos componentes de búsqueda (mutación o cruce) entre las islas deben ser usados. Otro problema de este esquema consiste en el máximo numero de individuos por isla que se pueden manejar ya que el máximo numero de hilos por bloque es limitado (1024 de acuerdo a la arquitectura de GPU actuales) [4].

Una idea para solucionar esta limitación es asociar una isla con muchos bloques de hilos [22]. Sin embargo, esto no puede ser fácilmente implementado en la práctica ya que los hilos trabajan de manera asíncrona y las sincronizaciones de hilos son locales para el mismo bloque.

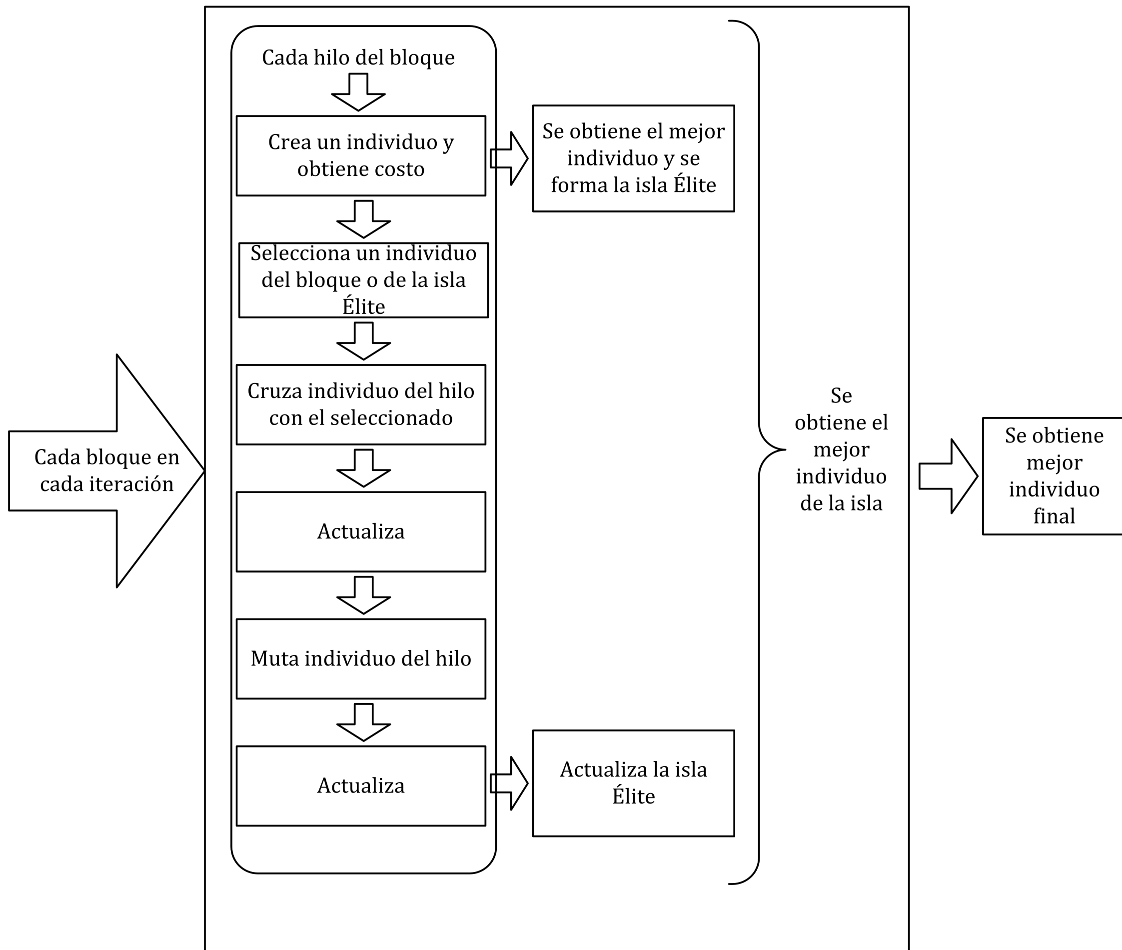


Fig. 4.7: Modelo de islas de AGP con isla Élite

Esto produciría errores de ejecución si la selección es hecha en dos individuos de un bloque diferente donde uno de los dos individuos posiblemente aun no ha sido evaluado.

El Algoritmo 3 muestra el pseudocódigo para el Algoritmo Genético Paralelo con isla Élite implementado en CUDA. Como todas las operaciones del Algoritmo Genético se hacen dentro de la GPU, cada hilo en el GPU crea un individuo, al cual posteriormente se trata de mejorar con las operaciones de cruce y mutación. Los individuos que se crearon con hilos del mismo bloque, son individuos que pertenecen a la misma isla, de tal manera que en cada isla (bloque) se hace evolucionar a los individuos que pertenecen a la misma.

Cada hilo de la función kernel actualiza los parámetros de un individuo. La isla Élite, contiene el mejor individuo de cada una de las otras islas. Al momento de hacer cruce, cualquier hilo tiene acceso a la isla Élite, entonces, existe la posibilidad de que el individuo de cada hilo, pueda cruzarse con cualquiera de los individuos de la isla Élite, en búsqueda de mejores soluciones.

Se utiliza el algoritmo Greedy para generar a los individuos de la población inicial en cada bloque. El operador de cruce es por orden (OX) y el operador de mutación es el de inversión.

Algoritmo 3 Pseudocódigo del AGP con isla Élite

ENTRADA: las coordenadas (x,y) de las ciudades
SALIDA: mejor ruta obtenida con AGP
main()
asignación de memoria en el GPU
pc=probabilidad de cruce
pm=probabilidad de mutación
calcular distancias de ciudad a ciudad
copia de variables de CPU a GPU
bloque=ISLA
hilo=IND
initCurand<<<bloques,hilos>>>(parámetros)
Initpopulation<<<bloques,hilos>>>(parámetros)
while condiciónTerminar==FALSO **do**
  kernel_AG<<<bloques,hilos>>>(parámetros)
end while
copia de memoria de la IE del GPU a CPU
obtener mejor solución de IE

__global__ Initpopulation()
crea(IND)
evaluar(IND)
bestI=mejor IND de la ISLA
se crea la isla élite (IE)

__global__ kernel_AG()
selección(IND,IE)
Inext = cruce(IND,IE)
Evaluar(INDnext)
if costo(INDnext)<costo(INDnext) **then**
  IND = INDnext
end if
if pmut < pm **then**
  INDnext = mutatión(I)
end if
se obtiene bestI de la ISLA
se migra bestI a IE

Cada hilo selecciona un individuo de su propia isla y se cruza con el individuo que pertenece a ese mismo hilo. El individuo generado por cruce, es colocado en la nueva población. Si el individuo generado es mejor que el que pertenece al hilo, se reemplaza. Posteriormente, se genera un número aleatorio y si este número es menor que la probabilidad de mutación, se procede a hacer mutación del individuo que se obtuvo por cruce. Enseguida, el individuo producto de la mutación, es comparado con el individuo mutado. El mejor de ellos pasa a ser parte de la siguiente población.

En cada iteración, se actualiza el mejor individuo de cada isla y se copia a la isla Élite. El algoritmo se detiene hasta completar una cierta cantidad de iteraciones, se obtiene la mejor solución, y finalmente se reporta por el CPU.

Capítulo 5

Resultados

En este capítulo se presentan resultados de un Algoritmo Genético secuencial y de Algoritmos Genéticos Paralelos con isla Élite implementados tanto en un procesador multinúcleo como en un GPU. Se describe la implementación de un Algoritmo Genético secuencial para el Problema del Agente Viajero con 131 ciudades. Este algoritmo se utilizó para encontrar el tipo de cruce y mutación que proporcionen mejores resultados para el problema dado y que fueron elegidos para ser utilizados en las implementaciones paralelas.

Los problemas de prueba se tomaron de la biblioteca de Problemas del Agente Viajero (TSPLIB) disponibles en [46]. Estos problemas son del tipo de PAV geométrico donde las ciudades están representadas por puntos cuyas coordenadas son leídas de un archivo proporcionado en tal librería. El algoritmo calcula la distancia que existe entre cada par de ciudades la cual está determinada por

$$d(c_i, c_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}, \tag{5.1}$$

donde los puntos (x_i, y_i) y (x_j, y_j) son las coordenadas de las ciudades c_i y c_j respectivamente. En este trabajo se resuelve el Problema del Agente Viajero simétrico. Por lo tanto, la distancia $d(c_i, c_j) = d(c_j, c_i)$.

Se analiza el desempeño de los algoritmos implementados en un procesador multinúcleo y en una GPU tomando en cuenta la calidad de las soluciones y el tiempo necesario para obtenerlas.

En esta investigación llamamos solución óptima a la mejor que se reporta en la biblioteca TSPLIB para cada instancia del problema. El error de aproximación es

$$Error = \frac{MS - OLIB}{OLIB} \times 100, \tag{5.2}$$

donde MS es la mejor solución que arrojó el algoritmo propuesto y el $OLIB$ es la mejor solución considerada como óptima reportada en TSPLIB.

Tanto el Algoritmo Genético secuencial como el Algoritmo Genético Paralelo en procesa-

dor multinúcleo se ejecutaron en ambiente GNU/Linux y lenguaje de programación C en una computadora con un procesador intel i7-4770 de 3.4 GHz y 16 GBytes de memoria RAM. En el caso de la implementación en GPU, esta se hace en GNU/Linux con el mismo procesador antes descrito y el modelo de programación CUDA en un GPU *GeForce GTX 670* con 2 GByte de memoria global.

5.1. Solución al Problema del Agente Viajero con un Algoritmo Genético secuencial

El Algoritmo Genético simple, se muestra en el Algoritmo 1 en la sección 3.2. En base a él, se implementó un Algoritmo Genético Secuencial que se ejecuta por un solo hilo de ejecución del CPU, de tal manera que todos los pasos del mismo son ejecutados uno tras otro.

En las secciones 3.4.5 y 3.4.6 se mencionó que existen diferentes tipos de cruce y de mutación que pueden ser utilizados para resolver el Problema del Agente Viajero por medio de Algoritmos Genéticos. Con la intención de encontrar el tipo de cruce y mutación, así como la mejor combinación de estos que proporcionen mejores resultados, se hicieron implementaciones de un AG secuencial para el PAV con 131 ciudades, y se compararon los resultados proporcionados por dichos operadores. La solución óptima reportada en TSPLIB [46] para este problema es 564 unidades de longitud.

El algoritmo se ejecutó con diferente cantidad de individuos en la población inicial, así como para diferente número de iteraciones. Se hace evolucionar a los individuos de la población y se reporta la mejor solución obtenida. El tipo de cruce que se utilizó fué el cruce por orden (OX) y mutación por inversión con una probabilidad de 80 % y 10 % respectivamente. Cada ejecución se realizó 30 veces, se obtuvo el promedio y la mejor de las soluciones encontradas, así como el tiempo de ejecución. En la Tabla 5.1 se observa que con una población de 1000 individuos y 300000 iteraciones se aproximó más a la solución óptima, con una distancia de 580 unidades de longitud y el promedio más bajo.

Tabla 5.1: Cruce por orden (OX) y mutación por inversión.

Individuos	Iteraciones								
	100000			200000			300000		
	MS	Prom	T(s)	MS	Prom	T(s)	MS	Prom	T(s)
1000	588	595	3.48	586	594	6.85	580	592	10.22
2000	592	601	3.55	591	602	6.93	581	594	10.33
3000	591	604	3.65	587	598	7.01	592	598	10.37
4000	591	604	3.70	581	597	7.10	590	598	10.94

Tabla 5.2: AG secuencial con población inicial de 1000 individuos.

Cruce	Mutación	Iteraciones								
		100000			200000			300000		
		MS	Prom	T(s)	MS	Prom	T(s)	MS	Prom	T(s)
OX	Inversión	588	595	3.48	586	594	6.85	580	592	10.22
OX	Inserción	610	619	2.90	599	604	5.80	596	599	10.30
PMX	Inversión	615	633	1.07	612	621	3.40	600	604	4.93

Debido a que se obtuvieron los mejores resultados con una población inicial de 1000 individuos, se estableció este dato como parámetro para probar otro tipo de cruce y de mutación.

Los tipos de cruce por orden y por emparejamiento parcial, así como mutación por inversión y por inserción fueron los operadores evaluados y cuyo resultado se muestra en la Tabla 5.2. La solución más cercana a la óptima así como un mejor promedio se obtuvieron con el operador de cruce por orden (OX) en combinación con el operador de mutación por inversión. Como puede verse, esta combinación de operadores genéticos tardan más tiempo que el cruce PMX combinado con mutación por inversión. Sin embargo, se puede observar que se encuentran soluciones más cercanas a la óptima que se reporta en TSPLIB, y por esta razón, se optó por elegir estos operadores para ser utilizados en las implementaciones paralelas.

5.2. Solución al Problema del Agente Viajero con Algoritmos Genéticos Paralelos en un procesador multinúcleo

El Algoritmo Genético Paralelo con isla Élite se programó con hilos POSIX [47] en un procesador multinúcleo. Aumentar el número de hilos de ejecución permite abarcar una mayor cantidad de posibles soluciones. Cada hilo genera una población inicial con el algoritmo Greedy y se crea la isla Élite con el mejor individuo de cada una de las islas. Para seleccionar los individuos se utiliza el tipo de seleccón por torneo y se hace evolucionar a la población con el operador de cruce por orden (OX) y el operador de mutación por inversión. Cuando en una isla se encuentra un mejor individuo, se evalúa si este nuevo individuo es mejor que el individuo presente en la isla Élite identificado como el mejor de dicha isla. En caso de que así sea, se actualiza el individuo de la isla Élite. Después de cierto número de iteraciones, se obtiene el mejor individuo presente en la isla Élite y se reporta como el mejor individuo obtenido.

Los resultados para las instancias del PAV implementados en un procesador multinúcleo, se presentan en la Tabla 5.3. En la primer columna se muestran los problemas que fueron utilizados. Los tres últimos dígitos del nombre de cada problema indican el número de ciudades del que

Tabla 5.3: Soluciones obtenidas en un procesador multinúcleo

Problema	Ind	Islas	Iter	OLIB	MS	Peor	Prom	Error %	T(s)
XQF131	2000	20	10000	564	573	580	575	1.60	0.72
XQG237	3000	40	30000	1019	1030	1060	1043	1.08	8.70
BCL380	20000	40	500000	1621	1657	1679	1672	2.22	291.90
XQL662	10000	40	500000	2513	2617	2690	2653	4.14	1153.10
DKG813	20000	30	500000	3199	3340	3397	3378	4.41	1343.87
XIT1083	10000	20	500000	3558	3791	3857	3817	6.55	1534.48

consta cada uno de ellos. La calidad de las soluciones encontradas por medio de Algoritmos Genéticos depende de la combinación de los parámetros del mismo. Los parámetros modificados fueron la cantidad de individuos de la población, el número de islas y la cantidad de iteraciones. La probabilidad de cruce y mutación son 80 % y 10 % respectivamente. De la misma manera que el AG secuencial, el AGP se ejecutó 30 veces y se obtuvo el promedio, se presenta la mejor de las soluciones encontradas y el tiempo de ejecución.

Cuando se incrementa el número de ciudades del Problema del Agente Viajero se vuelve más difícil acercarse más a la solución óptima. Se puede observar en la Tabla 5.3 que para el problema XQF131 se tiene un error mayor que para XQG237. Esto se debe a que el algoritmo se estanca en un óptimo local para el problema XQF131, del que ya no sale aunque se aumente el número de iteraciones.

En la Tabla 5.4 se hace una comparación entre los resultados de la ejecución del AG secuencial y el AGP en un procesador multinúcleo para el PAV con 131 ciudades. Se observa que gracias a la paralelización del algoritmo se obtuvo una mejor aproximación a la solución óptima reportada en [46]. Asimismo, dicha mejora se obtuvo con menos iteraciones en comparación con el AG secuencial, y por lo cual se disminuye el tiempo de ejecución.

Se calculó la desviación estándar para 10, 20 y 30 ejecuciones del algoritmo, cuyo valor se muestra en la Tabla 5.5. Se concluye que con 10 ejecuciones es suficiente para obtener los resultados mostrados en la Tabla 5.3 ya que el valor de la desviación estándar no presenta un cambio considerable para 20 y 30 ejecuciones.

Tabla 5.4: AG secuencial vs AGPE en un procesador multinúcleo

AG	MS	Prom	T(s)	Iteraciones
Secuencial	580	592	10.22	300000
Paralelo	573	575	0.72	10000

Tabla 5.5: Desviación estándar de los resultados en un procesador multinúcleo

Problema	# de ejecuciones		
	10	20	30
XQF131	2.34	2.20	2.06
XQG237	8.61	7.41	6.70
BCL380	7.68	7.30	6.71
XQL662	20.9	20.45	19.41
DKG813	15.52	16.66	16.33
XIT1083	17.95	17.21	17.25

5.3. Solución al Problema del Agente Viajero con Algoritmos Genéticos Paralelos en un GPU

En esta sección se presentan los resultados obtenidos de aplicar el Algoritmo Genético Paralelo con isla Élite en un procesador gráfico. Se resolvieron las mismas instancias del PAV que con el algoritmo implementado en el procesador multinúcleo. A cada isla se le asigna un bloque de la GPU y cada individuo de la isla es mapeado en un hilo del bloque y se hace evolucionar a través de los operadores genéticos.

Se crea la población inicial con el algoritmo Greedy. La isla Élite es creada después de la población inicial. Se utiliza selección por torneo, cruce por orden y mutación por inversión. El porcentaje de cruce es de 80 % y el porcentaje de mutación de 10 %. Cada generación, cada hilo de ejecución crea un individuo y un hilo de cada bloque identifica al mejor individuo de su isla correspondiente y lo envía a la isla Élite. Despueés de ejecutarse un número determinado de iteraciones, se obtiene el mejor individuo presente en la isla Élite y se toma como el mejor individuo obtenido.

Cuando el tamaño del problema crece, aumenta la cantidad de posibles soluciones que podemos obtener [7], entonces, se debe aumentar el tamaño de la población inicial y como consecuencia, son necesarios más recursos de cómputo así como mayor tiempo para encontrar soluciones más aproximadas a las óptimas.

En la Tabla 5.6 se presentan los resultados del algoritmo implementado en una GPU. Para cada una de las instancias que se resuelven, se utilizó diferente número de individuos e islas debido a que son problemas de diferente número de ciudades. Se observa que al aumentar el tamaño del problema, también aumenta el error con respecto a la solución óptima, excepto para el problema XQF131. En el caso de este problema, con los parámetros definidos no se obtuvo una mejor solución que la obtenida con el procesador multinúcleo. Sin embargo, el promedio si disminuye, incluso la desviación estándar para este problema se hace 0 como se muestra en la Tabla 5.7.

Tabla 5.6: Soluciones obtenidas para instancias del TSPLIB con el algoritmo implementado en un GPU

Problema	Ind	Islas	Iter	OLIB	MS	Peor	Prom	Error %	T(s)
XQF131	512	20	500	564	573	573	573	1.60	6.33
XQG237	512	20	1000	1019	1028	1037	1033	0.88	30.41
BCL380	512	20	1000	1621	1643	1665	1657	0.99	51.73
XQL662	1024	40	1000	2513	2609	2647	2635	3.90	893.24
DKG813	1024	30	1000	3199	3325	3355	3338	3.94	895.25
XIT1083	1024	22	4000	3558	3745	3816	3788	5.26	3124.8

Tabla 5.7: Desviación estándar de los resultados en un GPU

Problema	# de ejecuciones		
	10	20	30
XQF131	0	0	0
XQG237	3.44	3.38	3.08
BCL380	10.49	9.28	8.32
XQL662	17.51	16.44	15.5
DKG813	12.35	11.93	11.24
XIT1083	26.11	26.25	23.73

Se calculó la desviación estándar para 10, 20 y 30 ejecuciones del algoritmo, cuyo valor se muestra en la Tabla 5.7. Se concluye que con 10 ejecuciones es suficiente para obtener los resultados mostrados en la Tabla 5.6 ya que el valor de la desviación estándar no presenta un cambio considerable para 20 y 30 ejecuciones. Lo que puede variar si se ejecuta un mayor número de veces el algoritmo, es el promedio, ya que la mejor y peor solución reportadas, siempre se encuentran con tan solo hacer 10 ejecuciones.

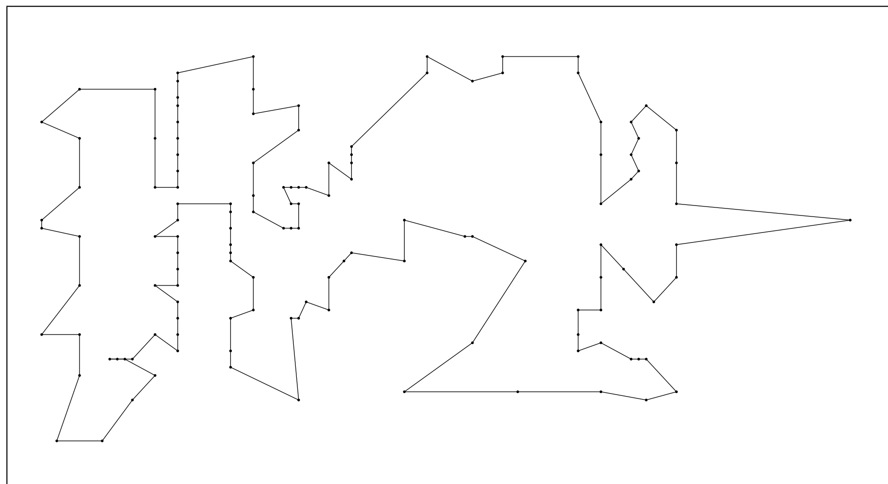


Fig. 5.1: Ruta de la mejor solución obtenida para 131 ciudades

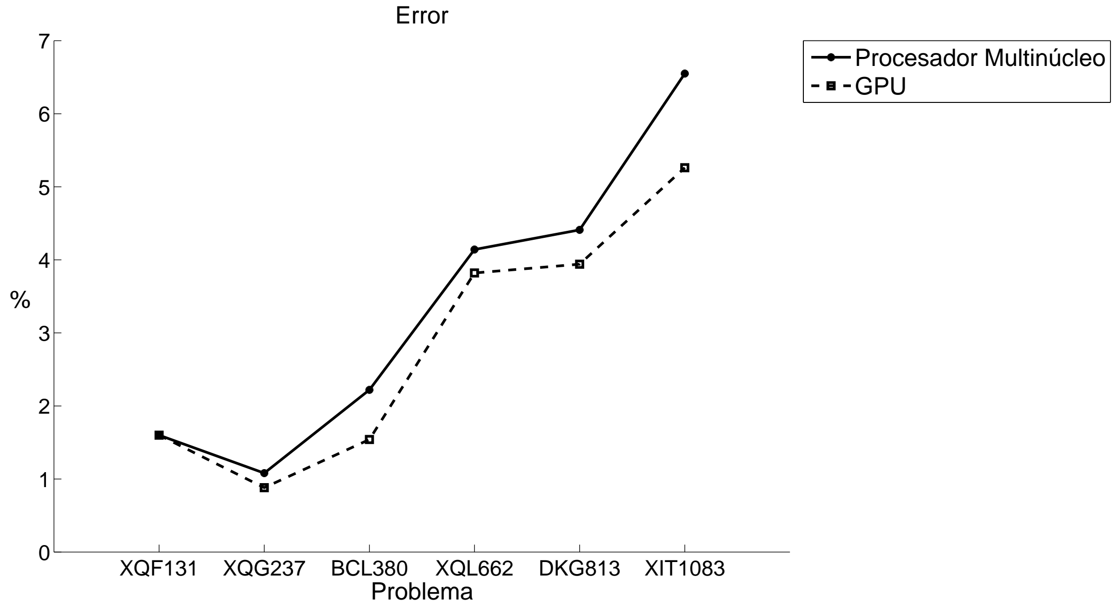


Fig. 5.2: Error de las implementaciones en un procesador multinúcleo y en una GPU

Cabe señalar que para el problema XQF131, en dos ocasiones se obtuvo una solución de 565 unidades de longitud. La Fig. 5.1 muestra la ruta que corresponde a tal solución, donde cada punto que se observa representa una ciudad de destino. Éste resultado se obtuvo con los mismos parámetros presentados en la Tabla 5.6, la diferencia es que se utilizó una probabilidad de mutación de 100 %. Sin embargo al ejecutar más veces el algoritmo, no se presentó de nuevo dicha solución. Ante esta situación se puede pensar en la posibilidad de que el algoritmo mejore si se modifican la probabilidad de cruce y mutación, tanto para este problema, como para los demás que aquí se resuelven.

Para los problemas XQG237, BCL380, XQL662, DKG813 y XIT1083, cuyas mejores soluciones obtenidas con el algoritmo implementado en GPU se muestran en la Fig. 5.3, Fig. 5.4, Fig. 5.5, Fig. 5.6 y Fig. 5.7 respectivamente, se obtuvieron soluciones más cercanas a la óptima en comparación con la implementación en procesador multinúcleo. En el caso del tiempo, se observa que para el problema XQF131 es mayor en la implementación en un GPU. Copiar datos del CPU al GPU y el gran número de veces que se accede a la memoria global del GPU, pueden ser los factores que influyen en tal resultado y que por el tamaño del problema no se alcance a presentar una mejora en el tiempo en comparación con la implementación el procesador multiúcleo.

Para los problemas XQG237 y XIT1083 el tiempo de ejecución en GPU es mayor que en la implementación en el procesador multinúcleo. Sin embargo, en estos casos este incremento se justifica por el hecho de que en GPU se obtuvieron soluciones más cercanas a las óptimas.

En la Fig. 5.2 se muestra el error que se obtiene con la implementación del AGPE en un

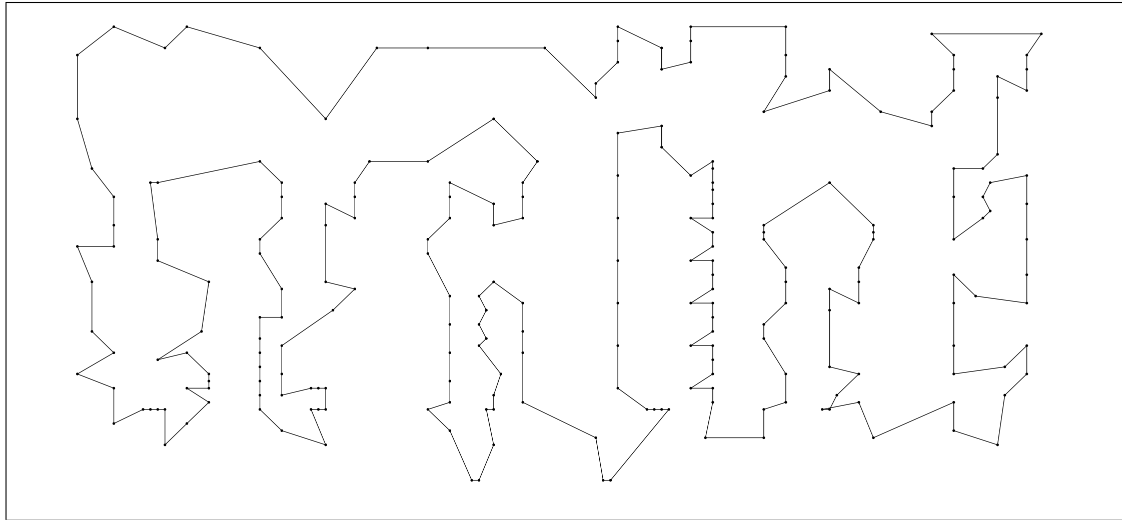


Fig. 5.3: Ruta de la mejor solución obtenida para 237 ciudades

procesador multinúcleo y el mismo algoritmo implementado en una GPU para las seis instancias del PAV que se resuelven en este trabajo de investigación.

Los individuos de cada población así como su valor de aptitud correspondiente, están almacenados en la memoria global del GPU. El acceso a dicha memoria, es más lento que el acceso a la memoria compartida. Se pensó en la posibilidad de disminuir los tiempos de ejecución del algoritmo y guardar la población de cada isla en la memoria compartida de GPU. Sin embargo, como se explicó anteriormente, el tamaño de la memoria compartida limita el tamaño de cada población, es decir, solo se podrían manejar unos pocos individuos dentro de una población, lo que resulta impráctico para hacer una buena búsqueda global [26].

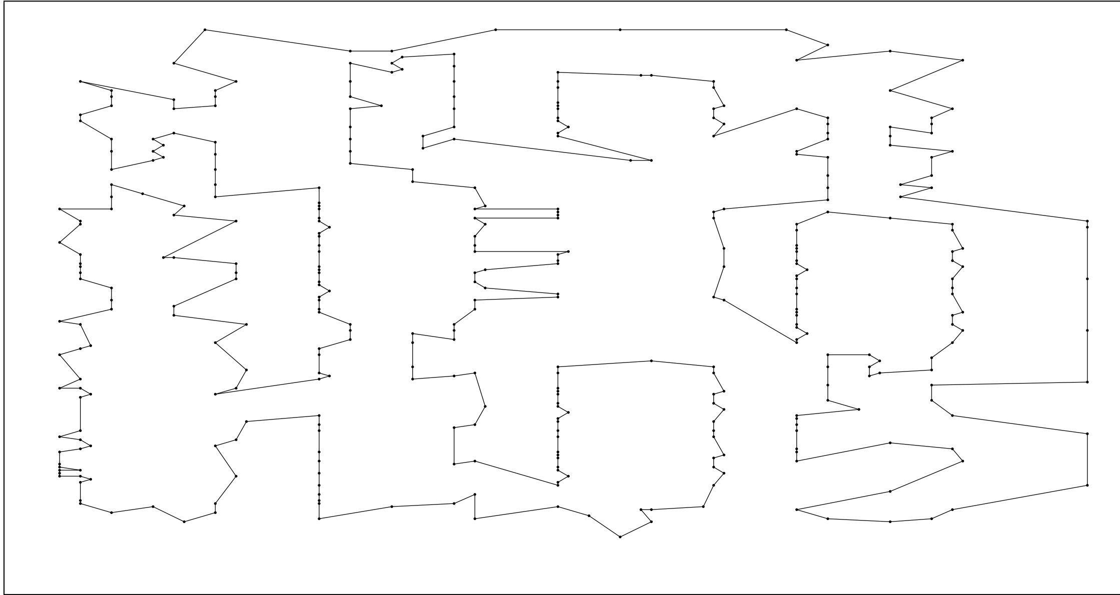


Fig. 5.4: Ruta de la mejor solución obtenida para 380 ciudades

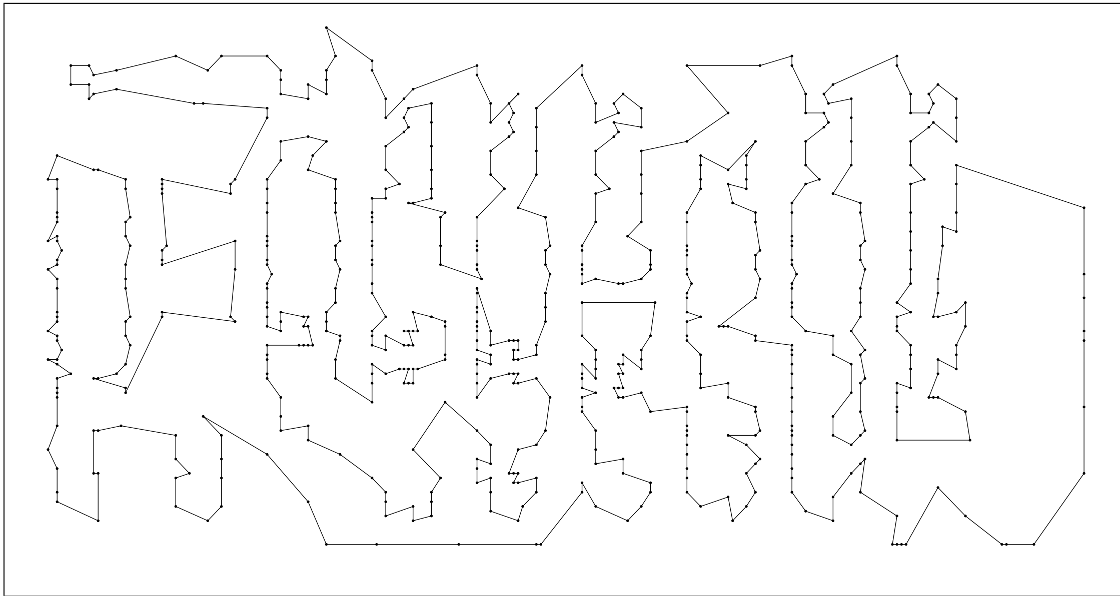


Fig. 5.5: Ruta de la mejor solución obtenida para 662 ciudades

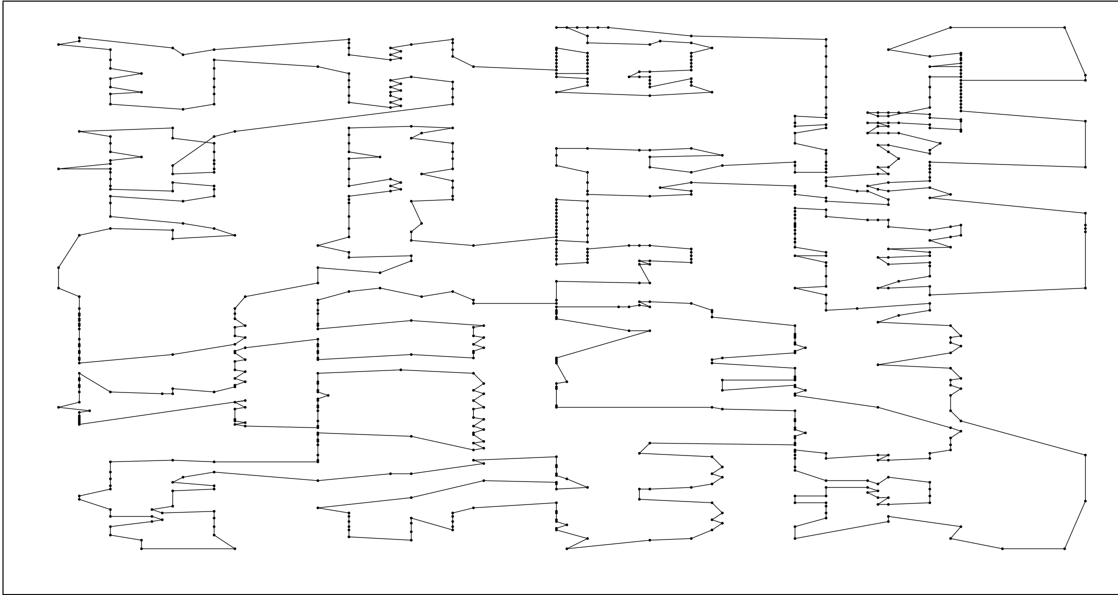


Fig. 5.6: Ruta de la mejor solución obtenida para 813 ciudades

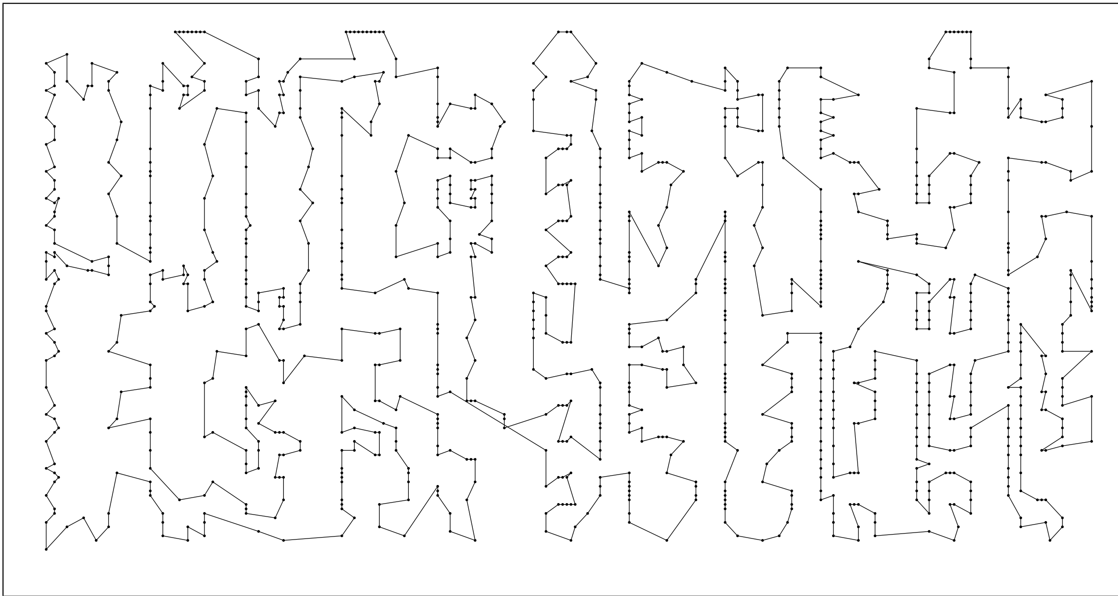


Fig. 5.7: Ruta de la mejor solución obtenida para 1083 ciudades

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se implementaron Algoritmos Genéticos Paralelos para solucionar seis instancias del Problema del Agente Viajero disponibles en la librería TSPLIB. Los algoritmos se implementaron en un procesador multinúcleo y en un GPU.

El algoritmo propuesto es llamado Algoritmo Genético Paralelo con isla Élite. Este no requiere de la migración de individuos entre las islas como el caso del modelo de islas de Algoritmos Genéticos Paralelos. Tiene como característica particular que se crea una isla a la que se denomina Élite donde se encuentra el mejor individuo de cada una de las islas. Los individuos del resto de las islas pueden cruzarse con los individuos de la isla Élite en búsqueda de mejores soluciones.

En un procesador multinúcleo, cada hilo POSIX ejecuta un Algoritmo Genético. El tiempo de ejecución del algoritmo en comparación con su ejecución en secuencial, se vé mejorado desde la ejecución de dos hilos hasta un número de hilos igual al número de núcleos que contiene el procesador. Utilizar más de 8 hilos ya no disminuye el tiempo de ejecución del algoritmo, pero si mejora la aproximación a la solución óptima. Esto se debe a las ventajas que ofrece el modelo de islas de Algoritmos Genéticos Paralelos en comparación de un Algoritmo Genético secuencial.

En la etapa de selección de un Algoritmo Genético, normalmente se eligen individuos al azar para cruzarlos y encontrar nuevos individuos. Por ser un proceso aleatorio, existe la posibilidad de que algunos individuos nunca sean utilizados para evolucionar a la población total.

En el caso de la implementación en un GPU, cada hilo de un bloque hace evolucionar a cada individuo de la población. Con esto, todos los individuos de la población son mejorados y participan en la evolución total de la misma. Gracias a esto, la calidad final de la población total resulta mejorada.

Para las instancias del Problema del Agente Viajero que en este trabajo de investigación se

utilizan, no es conveniente utilizar la memoria compartida del GPU para guardar a los individuos de cada población. Esto se debe a que la cantidad de dicha memoria no es suficiente para guardar la cantidad de individuos necesarios en cada isla.

Los resultados obtenidos indican que la implementación de este algoritmo en un GPU permite una mayor aproximación a la solución óptima que la implementación en un procesador multinúcleo.

6.2. Trabajo futuro

Como trabajo futuro para ampliar y mejorar el presente trabajo de investigación, se propone

- Resolver problemas con mayor cantidad de ciudades, así como implementar el Algoritmo Genético Paralelo con isla Élite a otros problemas de optimización.

- Implementar el algoritmo propuesto en un clúster de GPU. En este caso se pueden considerar las mismas instancias del Problema del Agente Viajero que aquí se trataron y utilizar más poblaciones distribuidas en más GPUs, o en su caso, resolver instancias con mayor número de ciudades.

- Encontrar una combinación de parámetros que nos proporcione mejores soluciones. Una opción sería variar la probabilidad de cruce y mutación ya que en este trabajo se mantuvieron sin variación.

- Utilizar un algoritmo de búsqueda local para tratar de mejorar las mejores soluciones que se obtuvieron. Se sugiere el algoritmo de búsqueda local 2-opt. Este algoritmo puede ser implementado después de haber obtenido la solución final con el Algoritmo Genético Paralelo con isla Élite, es decir, se buscaría mejorar localmente la mejor solución final encontrada. Otra idea es implementarlo cuando se encuentre una mejor solución por cruce, o en cada isla, directamente al mejor individuo de la misma. En estos dos casos, se espera que el tiempo de ejecución del algoritmo aumente considerablemente debido a que se harían múltiples ejecuciones del método 2-opt, pero se aumenta la posibilidad de encontrar mejores soluciones al explorar localmente los mejores individuos un mayor número de veces.

Referencias

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[2] T. Rauber and G. Runger, *Parallel Programming for Multicore and Cluster Systems.* Springer, 2010.

[3] J. Sanders and E. Kandrot, *CUDA by Example: An introduction to General-Purpose GPU Programming.* NVIDIA Corporation, 2010.

[4] NVIDIA, *CUDA C Programming Guide Version 5.5*, 2013.

[5] O. Goldreich, *Computational Complexity a Conceptual Perspective.* Cambridge, 2008.

[6] S. Arora and B. Barak, *Computational Complexity a Modern Approach.* Cambridge, 2009.

[7] G. Gutin and A. P. Punnen, *The Traveling Salesman Problem and Its Variations.* Kluwer Academic Publishers, 2004.

[8] J. H. Holland, *Adaption in Natural and Artificial Systems.* The MIT Press, 1992.

[9] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms.* Springer, 2008.

[10] G. Luque and E. Alba, *Parallel Genetic Algorithms.* Springer, 2011.

[11] M. Affenzeller, S. Winkler, S. Wagner, and A. Beham, *Genetic Algorithms and Programming.* Taylor and Francis Group, LLc., 2009.

[12] N. Fujimoto and S. Tsutsui, "A Highly-Parallel TSP Solver for a GPU Computing Platform," in *Numerical Methods and Applications*, I. Dimov, S. Dimova, and N. Kolkovska, Eds. Springer, 2011, pp. 264–271.

[13] H. J. Su Chen, Spencer Davis and A. Novobilski, "CUDA-based genetic algorithm on traveling salesman problem," in *Computer and Information Science*, R. Lee, Ed. Springer, 2011, pp. 241–252.

[14] K. Rocki and R. Suda, "Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012)*, 2012, pp. 705–706.

[15] N. Fujimoto and S. Tsutsui, "Parallelizing a genetic operator for GPUs," in *2013 IEEE Congress on Evolutionary Computation*, june 2013, pp. 1271–1277.

[16] S. Tsutsui and N. Fujimoto, "Solving quadratic assignment problems by genetic algorithms with GPU computation: A case study," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, Montréal Québec, Canada, july 2009, pp. 2523–2530.

[17] R. Arora, R. Tulshyan, and K. Deb, "Parallelization of binary and real-coded genetic algorithms on GPU using CUDA," in *2010 IEEE Congress on Evolutionary Computation (CEC 2010)*, july 2010, pp. 1–8.

[18] P. Krömer, J. Platos, V. Snásel, and A. Abraham, "A comparison of many-threaded differential evolution and genetic algorithms on CUDA," in *Third World Congress on Nature and Biologically Inspired Computing (NaBIC)*, october 2011, pp. 509–514.

[19] J. Li, X. Lv, and L. Liu, "A parallel genetic algorithm with GPU accelerated for large-scale mdvrp in emergency logistics," in *Proceedings of the 2011 14th IEEE International Conference on Computational Science and Engineering*, ser. CSE '11, 2011, pp. 602–605.

[20] M. Wahib, A. Munawar, M. Munetomo, and K. Akama, "Optimization of parallel genetic algorithms for nvidia GPUs," in *2011 IEEE Congress on Evolutionary Computation (CEC),*, june 2011, pp. 803–811.

[21] M. C. Feier, C. Lemnaru, and R. Potolea, "Solving NP-complete problems on the CUDA architecture using genetic algorithms," in *2011 10th International Symposium on Parallel and Distributed Computing (ISPDC)*, july 2011, pp. 278–281.

[22] J. Jaros, "Multi-GPU island-based genetic algorithm for solving the knapsack problem," *IEEE World Congress on Computational Intelligence*, pp. 1–8, 2012.

[23] M. Kuroda, K. Yamamori, M. Munetomo, M. Yasunaga, and I. Yoshihara, "A proposal for zoning crossover of hybrid genetic algorithms for large-scale traveling salesman problems," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010*, Barcelona, Spain, 18-23 July 2010, pp. 1–6.

[24] Z. Z. Zhang, W. H. Huang, J. J. Lin, and P. C. C. J. L. Wu, "A puzzle-based artificial chromosome genetic algorithm for the traveling salesman problem," in *International Conference on Technologies and Applications of Artificial Intelligence (TAAI 2011)*, November 2011, pp. 299–304.

[25] J. Zhao, Q. Liu, W. Wang, Z. Wei, and P. Shi, "A parallel immune algorithm for traveling salesman problem and its application on cold rolling scheduling," *Information Sciences*, vol. 181, no. 7, pp. 1212–1223, april 2011.

[26] J. Hofmann, S. Limmer, and D. Fey, "Performance investigations of genetic algorithms on graphics cards," *Swarm evol. comput.*, vol. 12, pp. 33–47, 2013.

[27] M. L. Wong and T. T. Wong, "Implementation of parallel genetic algorithms on graphics processing units," in *Intelligent and Evolutionary Systems*, ser. Studies in Computational Intelligence, vol. 187. Springer, 2009, pp. 197–216.

[28] T. Cormen, C. Leiserson, R. Rivest, C. Stein, C. Leiserson, R. Rivest, C. Stein, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[29] S. Climer and W. Zhang, "Take a walk and cluster genes: A tsp-based approach to optimal rearrangement clustering," in *21st International Conference on Machine Learning*, Banff, Alberta, Canada, 2004, pp. 169–176.

[30] R. Diestel, *Graph Theory*, 4th ed. Springer, 2010.

[31] X. Bin and Z. Zhongyi, *Graph Theory.* East China Normal University Press, 2010.

[32] H.-C. Kuo and C.-H. Lin, "A directed genetic algorithm for global optimization," *Elsevier*, 2012.

[33] P. Vidal and E. Alba, "A multi-GPU implementation of a cellular genetic algorithm," in *2010 IEEE Congress on Evolutionary Computation (CEC 2010).* Barcelona: IEEE, 18-23 July 2010, pp. 1–7.

[34] E. Alba, *Parallel Metaheuristics: A new class of Algorithms*, 2005.

[35] L. Araujo and C. Cervigón, *Algoritmos Evolutivos.* Alfaomega Ra-ma, 2009.

[36] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms.* Wiley-Interscience, 2004.

[37] E. C. Paz, *Efficient and Acurrate Parallel Genetic Algorithms*, D. E. Goldberg, Ed. Kluwer Academic Publishers, 2001.

[38] J. Li, X. Wang, R. He, and Z. Chi, "An Efficient Fine-grained Parallel Genetic Algorithm Based on GPU-Accelerated," in *Network and Parallel Computing Worshop, 2007.*, School of Electronic & Information Engineering, Dalian University of Technology, DaLian, China. Liaoning: IEEE, 18-21 Sept 2007, pp. 855–862.

[39] Y. Wei, Y. Hu, and K. Gu, "Parallel search strategies for tsps using a greedy genetic algorithm," in *Proceedings of the Third International Conference on Natural Computation - Volume 03*, ser. ICNC '07. IEEE Computer Society, 2007, pp. 786–790.

[40] Z. Wang, H. Duan, and X. Zhang, "An improved greedy genetic algorithm for solving travelling salesman problem," in *Proceedings of the 2009 Fifth International Conference on Natural Computation - Volume 05*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 374–378.

[41] S. Ganguly, S. Mukherjee, D. Basu, and S. Das, "A novel strategy adaptive genetic algorithm with greedy local search for the permutation flowshop scheduling problem," in *Proceedings of the Third international conference on Swarm, Evolutionary, and Memetic Computing*, ser. SEMCCO'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 687–696.

[42] J. Reinders and A. Stepanov, *Intel Threading Building Blocks Outfitting C++ for Multi-core ProcessorParallelism*. O'Reilly, July 2007.

[43] D. B. Kirk and W. mei W.Hwu, *In Praise of Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[44] R. Farber, *CUDA Application Desing and Development*. Elsevier, 2011.

[45] NVIDIA, *CUDA C best practices guide*. NVIDIA, 2013.

[46] TSPLIB, "Georgia institute of technology, actualización mayo del 2013, www.tsp.gatech.edu/data/index.html."

[47] S. Akhter and J. Roberts, *Multi-Core Programming Increasing Performance through Software Multi-threading*, D. J. Clark, Ed. Intel Press, 2006.