Algoritmo Metaheurístico paralelo usando CUDA para solucionar problemas 0/1 Knapsack basado en el Procedimiento de Búsqueda del Pescador

Hernán Guillermo Dulcey Moran

Johny Andrés Ortega Ruiz

Director: PhD. Carlos Alberto Cobos Lozada

Co-directora: PhD. Martha Eliana Mendoza Becerra

Universidad del Cauca

Facultad de Ingeniería Electrónica y Telecomunicaciones

Departamento de Sistemas

Grupo de I+D en Tecnologías de la Información (GTI)

Área de Interés: Sistemas Inteligentes

Popayán, octubre 2016

TABLA DE CONTENIDO

1. INTRODUCCIÓN 15
1.1 Planteamiento del Problema 15
1.2 Aportes 17
1.3 Objetivos 18
1.3.1 Objetivo general 18
1.3.2 Objetivos específicos 18
1.4 Resultados obtenidos 19
2. CONTEXTO TEÓRICO Y ESTADO DEL ARTE 21
2.1 Contexto teórico 21
2.2 Estado del Arte 22
2.2.1 Algoritmos basados en búsqueda armónica 22
2.2.2 Algoritmos basados en optimización por enjambre de partículas 24
2.2.3 Otros enfoques de solución 25
2.2.4 Búsqueda armónica binaria simplificada (SBHS) 25
2.2.5 Competencia de liga de fútbol (SLC) 26
2.2.6 Saltos de ranas revueltas (MDSFL) 27
2.2.7 Procedimiento de Búsqueda del Pescador (FSP) 28
2.2.8 Algoritmo genético de pares monógamos (MopGA) 29
2.2.9 Algoritmo de búsqueda discreta gravitacional 30
2.2.10 Grado voraz y expectativa de eficiencia 30
3. PROCEDIMIENTO DE BÚSQUEDA DEL PESCADOR 31
3.1 SBFSP para resolver el problema 0/1 knapsack 31
3.1.1 Parámetros 34
3.1.1.1 Número de puntos de captura (N) 35
3.1.1.2 Número de nudos por punto de captura (M) 35
3.1.1.3 Probabilidad del mejor (p_m), iteraciones máximas (iMáx) y porcentaje de bits (p_b) 36
3.1.2 Procedimientos de reparación 37

3.1.2.1 Procedimiento de reparación simple 37
3.1.2.2 Procedimientos de reparación basados en densidad y valor 38
3.1.3 Mutación 41
3.1.3.1 Cruce uniforme medio (HUX) 41
3.1.3.2 Encendiendo bits 42
3.1.4 Diversidad 42
3.2 Computación de propósito general sobre la GPU 42
3.2.1 Kernels, grids, blocks y threads 44
3.2.2 Streaming Multiprocessors y warps 44
3.3 PBFSPG para resolver el problema 0/1 knapsack 46
3.3.1 Parámetros 47
3.3.2 Procedimiento de reparación 47
3.3.3 Algoritmo paralelo usando la GPU 47
3.3.3.1 Inicialización de los puntos de captura 47
3.3.3.2 Actualización del mejor punto de captura 47
3.3.3.3 Lanzamiento de la red de pesca 48
3.3.3.4 Actualizar los puntos de captura 49
3.3.3.5 Buscar el punto de captura xs similar a xg 49
3.3.3 Versiones paralelas 49
4. METAHURÍSTICAS PARALELAS USANDO LA GPU PARA RESOLVER EL PROBLEMA 0/1 KNAPSACK 53
4.1 Enfoques de solución para las metaheurísticas que solucionan el problema 0/1 Knapsack 54
4.1.2 Soluciones basadas en penalización 54
4.1.3 Soluciones basadas en reparación 55
4.2 Saltos de ranas revueltas usando la GPU (PMDSFLG) 55
4.2.1 Parámetros 55
4.2.2 Procedimiento de reparación 55
4.2.3 Algoritmo paralelo usando la GPU 56
4.2.3.1 Inicialización de la población 57

4.2.3.2 División de la población en memeplexes 57
4.2.3.3 Búsqueda local 58
4.2.3.4 Mutación de la población 59
4.2.3.5 Actualizar la mejor rana global 59
4.2.4 Versiones paralelas 59
4.3 Algoritmo genético de pares monógamos usando la GPU (PMopGAG) 60
4.3.1 Parámetros 60
4.3.2 Función objetivo 60
4.3.3 Algoritmo paralelo usando la GPU 61
4.3.3.1 Inicialización de la población 62
4.3.3.2 Creación de las familias 62
4.3.3.3 Infidelidad 62
4.3.3.4 Creación de los hijos 62
4.3.3.5 Reconciliación 63
4.3.3.6 Competencia de la descendencia 63
4.3.3.7 Sobrevivientes de la competencia de la descendencia 64
4.3.3.8 Complejo de Edipo 65
4.3.4 Versiones paralelas 65
4.4 Búsqueda armónica binaria simplificada usando la GPU (PSBHSG) 65
4.4.1 Parámetros 66
4.4.2 Procedimiento de reparación 67
4.4.3 Algoritmo paralelo usando la GPU 67
4.4.3.1 Inicialización de las memorias armónicas 67
4.4.3.2 Proceso de improvisación 67
4.4.3.3 Actualización de las memorias armónicas 68
4.4.3.4 Actualización de la mejor armonía 68
4.4.4 Versiones paralelas 69

4.5 Competencia de liga de fútbol usando la GPU (PSLCG) 70
4.5.1 Función objetivo 71
4.5.2 Parámetros 71
4.5.3 Algoritmo paralelo usando la GPU 72
4.5.3.1 Inicialización de los jugadores 72
4.5.3.2 Ordenar jugadores 72
4.5.3.3 Fijar los jugadores estrella 72
4.5.3.4 Calcular la potencia del equipo 73
4.5.3.5 Realizar jornada 73
4.5.3.6 Aplicar operadores de imitación y provocación 74
4.5.3.7 Actualizar el jugador súper estrella 75
4.5.4 Versiones paralelas 75
5. EXPERIMENTACIÓN 77
5.1 Instancias del problema 0/1 knapsack 77
5.1.1 Tipos de Instancias del problema 0/1 knapsack 78
5.2 Afinamiento de parámetros 80
5.2.1 CAs aplicados al afinamiento de parámetros 80
5.2.1.1 Ejemplo de afinamiento de parámetros para PSBHSG 81
5.3 Resultados y discusión 84
5.3.1 Criterios de evaluación de las metaheurísticas 84
5.3.2 Instancias de prueba 85
5.3.3 Resultados en el afinamiento de parámetros 85
5.3.4 Resultados de las metaheurísticas secuenciales 86
5.3.5 Resultados de las metaheurísticas paralelas usando la GPU 90
5.3.6 Análisis de la aceleración de PBFSPG 95
6. CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO 99
6.1 Conclusiones 99
6.2 Trabajo futuro 102
6.3 Recomendaciones 103

BIBLIOGRAFÍA 105

LISTA DE FIGURAS

Figura 3-1 Evolución de un punto de captura en SBFSP 32
Figura 3-2 Algoritmo SBFSP 34
Figura 3-3 Procedimiento de reparación basado en S_x 40
Figura 3-4 Etapa dos GPU 41
Figura 3-5 Ejecución del kernel sumaMatrices usando CUDA C/C++ 43
Figura 3-6 Evolución de los puntos de captura en PBFSPG 46
Figura 3-7 Algoritmo PBFSPG 48
Figura 4-1 Metaheurísticas para el problema 0/1 Knapsack 53
Figura 4-2 MDSFL usando la GPU 56
Figura 4-3 Algoritmo paralelo PMDSFLG 57
Figura 4-4 Estado de las nuevas familias 61
Figura 4-5 Estado de las familias para la iteración t + 1 63
Figura 4-6 Competencia de la descendencia 63
Figura 4-7 Algoritmo paralelo PMopGAG 64
Figura 4-8 SBHS usando la GPU, iteración t 66
Figura 4-9 Algoritmo de improvisación 68
Figura 4-10 Algoritmo paralelo PSBHSG 69
Figura 4-11 Primera ronda (ida) de la liga 70
Figura 4-12 Definición del ganador de un encuentro 73
Figura 4-13 Algoritmo paralelo PSLCG 74
Figura 5-1 Instancias no correlacionadas 77
Figura 5-2 Instancias débilmente correlacionadas 78
Figura 5-3 Instancias fuertemente correlacionadas 79
Figura 5-4 Speed-up PBFSPG instancias no correlacionadas 96
Figura 5-5 Speed-up PBFPSG instancias débilmente correlacionadas 97
Figura 5-6 Speed-up PBFSPG instancias fuertemente correlacionadas 98

LISTA DE TABLAS

Tabla 5-1 Alfabeto para los parámetros de PSBHSG 82
Tabla 5-2 CA y Casos de pruebas para PSBHSG 83
Tabla 5-3 Parámetros metaheurísticas secuenciales 85
Tabla 5-4 Parámetros metaheurísticas paralelas 85
Tabla 5-5 Resultados secuenciales para el criterio valor óptimo para instancias no correlacionadas 87
Tabla 5-6 Resultados secuenciales para el criterio valor óptimo para instancias débilmente correlacionadas 88
Tabla 5-7 Resultados secuenciales para el criterio valor óptimo para instancias fuertemente correlacionadas 89
Tabla 5-8 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias no correlacionadas 91
Tabla 5-9 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias débilmente correlacionadas 92
Tabla 5-10 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias fuertemente correlacionadas 93
Tabla 5-11 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias difíciles 94
Tabla 5-12 Speed-up de SBFSP a PBFSPG 95

LISTA DE ANEXOS

Anexo A (Digital) Framework PMG_01KP 109

Anexo B (Digital) Archivos Excel con los resultados completos del ajuste de parámetros de las metaheurísticas paralelas usando la GPU, experimentos de las metaheurísticas secuenciales y experimentos de la metaheurísticas paralelas usando la GPU 111

Anexo C (Digital) Artículo “A Binary Fisherman Search Procedure for the 0/1 Knapsack Problem” 113

Anexo D (Digital) Artículo “Procedimiento de Búsqueda del Pescador Binario Paralelo usando la GPU para resolver el problema 0/1 knapsack para grandes dimensiones” 115

1. INTRODUCCIÓN

1.1 Planteamiento del Problema

Los problemas 0/1 Knapsack son problemas en los que se busca incluir en una mochila (knapsack) de capacidad limitada un grupo de objetos (de una lista de objetos disponibles) con el mayor valor acumulado posible y sin exceder la capacidad de la mochila. Este problema tiene varias aplicaciones en el mundo real, por ejemplo, para distribución de mercancías, distribución de recursos de capital, inversiones, selección de proyectos, entre otros [1, 2].

El problema 0/1 Knapsack puede ser abordado de forma lineal por programación dinámica [3] o en forma exhaustiva (evaluando todas las posibles combinaciones factibles), pero usando estos enfoques, a medida que la cantidad de objetos se incrementa o la capacidad de la mochila se incremente, el costo computacional es prohibitivo e incluso inviable [1].

Los algoritmos metaheurísticos, proveen una solución a este problema, guiando el proceso de búsqueda a mejores combinaciones y convergiendo más rápido a una buena solución con un límite de tiempo de ejecución. Recientemente varios algoritmos se han propuesto para encontrar la mejor solución, entre ellos se han encontrado cuatro (4) que actualmente están a la vanguardia (Búsqueda armónica binaria simplificada, Competencia de Liga de Fútbol, Saltos de ranas revueltas y Algoritmo genético de pares monógamos), pero entre ellos no se ha definido cuál es el mejor [1, 4-6]. Además, basado en el No-Free Lunch Theorem (NFLT) [7] no se pude decir cuál es la mejor metaheurística para resolver este problema en concreto hasta que todas ellas no hayan sido estudiadas y comparadas experimentalmente.

Por otro lado, el problema 0/1 Knapsack tiene el potencial de ser infinitamente complejo, puesto que el número de objetos que pueden considerarse para ingresar en la mochila sigue incrementándose con la aparición de nuevos problemas y aplicaciones del mundo real [1, 8]. Para manejar esta complejidad, se está haciendo uso del computación en paralelo, con el objetivo de acortar los tiempos de cómputo para este tipo de problemas [9, 10]. Existen supercomputadores con grandes cantidades de núcleos, que actualmente se usan para resolver diversos problemas, pero es un hardware muy costoso y no accesible para todas las organizaciones.

Una solución mucho más barata para el cómputo paralelo, es la Arquitectura Unificada de Dispositivos de Cómputo (Compute Unified Device Architecture, CUDA), la cual hace uso de las tarjetas gráficas Nvidia, para procesar tareas simples de forma paralela. Una de las ventajas que tiene usar estos dispositivos, es que son ampliamente ofertados en el mercado puesto que son usados para videojuegos y ese mercado se extiende rápidamente por todo el mundo, además que su rango de precios es accesible a la mayoría de las organizaciones y personas [9, 10].

Teniendo en cuenta el NFLT y que el Procedimiento de Búsqueda del Pescador (Fisherman Search Procedure, FSP) [2] es una metaheurística sencilla, fácil de paralelizar, que reporta resultados muy competitivos en problemas de optimización continua, que puede ser adaptada a problemas discretos y binarios, y que además el Grupo de I+D en Tecnologías de la Información (GTI) ya ha trabajado en otros problemas de optimización complejos, como en la generación automática de resúmenes [11], un problema binario y de alta dimensionalidad, en este proyecto se consideró apropiado usar FSP para resolver el problema 0/1 knapsack.

FSP halla las nuevas soluciones usando una combinación de búsqueda guiada y búsqueda local. En [2] se proponen nuevas características que extienden FSP, entre ellas, se plantea un procedimiento de pesca cooperativa mediante el uso de cómputo paralelo, característica que se desarrolla en el presente trabajo.

A partir de lo anteriormente dicho, en este proyecto se plantearon las siguientes preguntas de investigación ¿Cuál de los cuatro mejores algoritmos actuales (Búsqueda armónica binaria simplificada, Competencia de Liga de Fútbol, Saltos de ranas revueltas y Algoritmo genético de pares monógamos) es el mejor algoritmo para resolver el problema 0/1 Knapsack?, ¿Cuál es el mejor de los cuatro algoritmos cuando se ejecuta en forma paralela usando CUDA? Y finalmente, ¿Una propuesta basada en el Procedimiento de Búsqueda del Pescador puede obtener mejores resultados en ejecución secuencial o paralela frente a los cuatro algoritmos estudiados?

1.2 Aportes

Desde la perspectiva de investigación las contribuciones de este proyecto van encaminadas a generar nuevo conocimiento, centrado en definir cuál de los algoritmos en el estado del arte realmente es el mejor en la solución del problema 0/1 knapsack en forma secuencial y paralela.

Además, se propone un nuevo algoritmo metaheurístico basado en la metaheurística del Procedimiento de Búsqueda del Pescador para la solución de dicho problema y se evaluara si es o no mejor que lo presentado en el estado del arte, lo que también es nuevo conocimiento en el área de los sistemas inteligentes. Este último aporte se hace teniendo en cuenta que en la actualidad no se encontró ningún reporte definitivo del tema en las bases de datos de IEEE, ScienceDirect y Springer Link y el NFLT. Este nuevo conocimiento espera ser de tipo exploratorio y descriptivo, respondiendo a preguntas como: ¿Es posible lograr mejores soluciones a los problemas 0/1 knapsack con el nuevo algoritmo metaheurístico propuesto basado en la metaheurística del Procedimiento de Búsqueda del Pescador, o no?, ¿Cuáles son las condiciones que hacen que el nuevo algoritmo metaheurístico propuesto basado en el Procedimiento de Búsqueda del Pescador mejore o empeore los resultados en

la solución de problemas 0/1 knapsack, en cuanto a encontrar la mejor solución de problemas en el menor tiempo posible?

Desde la perspectiva de innovación con este proyecto se implementará en forma secuencial y paralela el nuevo algoritmo propuesto y los cuatro (4) mejores del estado del arte para resolver problemas 0/1 knapsack, los cuales pueden luego ser usados en problemas concretos o reales de la industria.

1.3 Objetivos

A continuación, se presentan los objetivos aprobados en el anteproyecto de este trabajo de grado por parte del Consejo de Facultad de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca.

1.3.1 Objetivo general

Proponer un algoritmo Metaheurístico paralelo usando Arquitectura Unificada de Dispositivos de Cómputo (Compute Unified Device Architecture, CUDA) para solucionar problemas 0/1 Knapsack basado en el Procedimiento de Búsqueda del Pescador (Fisherman Search Procedure, FSP).

1.3.2 Objetivos específicos

- Realizar un estudio comparativo de los algoritmos Saltos de Rana Revueltas, Búsqueda Armónica Binaria Simplificada y Competencias de Ligas de Fútbol para resolver problemas 0/1 Knapsack en forma secuencial y paralela usando CUDA.
- Proponer un nuevo algoritmo metaheurístico para solucionar problemas 0/1 knapsack basado en el procedimiento de búsqueda del pescador con una

modificación en el proceso de búsqueda local usando diferentes esquemas de vecindario fijo y variable.

- Evaluar y comparar el nuevo algoritmo propuesto con los otros tres algoritmos previamente estudiados y comparados, en problemas 0/1 knapsack de diferente grado de dificultad usando tiempo y evaluaciones de función objetivo promedio para alcanzar la mejor respuesta conocida y mejor solución promedio obtenida cuando el algoritmo se ejecuta un número específico de evaluaciones de la función objetivo (o tiempo específico).

1.4 Resultados obtenidos

- El desarrollo de un Framework de pruebas denominado PMG_01KP (parallel metaheuristics using GPU for the 0/1 knapsack problem), donde están implementados los algoritmos del estado del arte (los cuatro algoritmos metaheurísticos seleccionados), además del algoritmo del pescador en sus respectivas versiones secuenciales y paralelas. Este Framework permite obtener información de cada prueba, como lo son: la tasa de éxito, mejor valor de función objetivo, peor valor de función objetivo, mediana, promedio y desviación estándar de los valores de función objetivo, mejor tiempo, peor tiempo, mediana, promedio y desviación estándar de los valores de tiempo, menor número de evaluaciones de la función objetivo, mayor número de evaluaciones de la función objetivo, mediana, promedio y desviación estándar del número de evaluaciones de la función objetivo.
- Los resultados de las pruebas hechas a los algoritmos en cuestión, con su respectiva comparación y análisis de los resultados.
- Monografía, el presente documento donde se presenta a modo de reporte final, la descripción de las metaheurísticas usadas para resolver problemas 0/1 knapsack, la propuesta del pescador, los resultados de las comparaciones de los algoritmos y el análisis de esos resultados. Además, se presentan las

conclusiones principales del trabajo y el trabajo futuro que el grupo de investigación espera realizar en el tema.

- Un artículo publicado en evento internacional con memorias indexadas A1 por el Publindex de Colciencias y Q3 en SJR. Este documento muestra el análisis de los resultados obtenidos por cuatro algoritmos metaheurísticos (Búsqueda armónica binaria simplificada, Competencia de Liga de Fútbol, Saltos de ranas revueltas y la primera propuesta del Procedimiento de Búsqueda del Pescador desarrollada en este trabajo) para la solución de problemas 0/1 knapsack con implementación secuencial. El artículo tiene la siguiente referencia: C. Cobos, H. Dulcey, J. Ortega, M. Mendoza, A. Ordoñez. A Binary Fisherman Search Procedure for the 0/1 Knapsack Problem. Advances in Artificial Intelligence: Proceedings of the 17th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2016. O. Luaces et al. (Eds.). Springer International Publishing Switzerland. pp. 447-457. Series: Lecture Notes in Computer Science (LNCS), Subseries: Lecture Notes in Artificial Intelligence (LNAI). Salamanca, Spain, September 14-16, 2016, vol. 9868. ISSN: 0302-9743 (Print) 1611-3349 (Online). Available online at http://link.springer.com/chapter/10.1007%2F978-3-319-44636-3_42.
- Un artículo final que se espera enviar a evaluación a revista internacional indexada JCR a finales de noviembre de 2016. Este documento se anexa a la presente monografía en español y contempla la comparación de los cuatro algoritmos del estado del arte junto con la propuesta basada en FSP en su implementación paralela usando CUDA.

2. CONTEXTO TEÓRICO Y ESTADO DEL ARTE

2.1 Contexto teórico

El objetivo del problema 0/1 Knapsack es maximizar el valor acumulado de los objetos empacados en una mochila (Knapsack) bajo la condición de que el volumen total correspondiente a los objetos no exceda la capacidad de volumen de la mochila. Matemáticamente, este problema puede expresarse según la Ecuación (2-1).

$$Max\ f(x) = \sum_{i=1}^{n} v_i \times x_i$$
$$sujeto\ a \sum_{i=1}^{n} w_i \times x_i \ \leq \ w_{máx} \qquad \textbf{(2-1)}$$
$$x_i \ \in \{0,1\}, i = 1,2,\ldots,n$$

Donde v_i es el beneficio o valor del objeto i y x_i es un valor entre 0 y 1 donde 1 significa que el objeto i va en la mochila y 0 en caso contrario, w_i es el volumen del objeto i y $w_{máx}$ es el volumen máximo de la mochila.

Los problemas 0/1 Knapsack usualmente no son diferenciables, son discontinuos, ruidosos (es decir que su forma gráfica es representada con mucho puntos sin una tendencia evidente), muy complejos (NP-completos), no lineales, con muchos óptimos locales y con restricciones complejas. Como las variables de decisión están restringidas a ser binarias (0 o 1), el espacio de variables está compuesto por un conjunto finito de puntos discretos en el cual se encuentra una solución óptima global. Por consiguiente, el método más directo e ingenuo de resolución consiste en enumerar exhaustivamente todas las soluciones y escoger una solución viable con el más alto valor como el óptimo global. Sin embargo, esto no es factible en la realidad puesto que la cantidad de objetos tiende a volverse cada vez más grande y el tiempo de solución sería muy alto, meses o años dependiendo del tamaño del problema.

Mientras tanto, los métodos tradicionales, como la programación dinámica [3] y la ramificación y poda [12], sufren del mismo problema. Para evitar esto, cada vez más investigadores centran su atención en el uso de algoritmos metaheurísticos o métodos de optimización de caja negra para resolver problemas genéricos, que usan aleatoriedad, estrategias de búsqueda global y optimización local, además en algunos casos incluye conocimiento sobre el problema, y generalmente imitan fenómenos naturales (mayormente bioinspirados) específicos para resolver diversos problemas de optimización complejos.

En las últimas décadas, una variedad de algoritmos de optimización metaheurísticos, se han desarrollado, entre ellos: algoritmos genéticos (Genetic Algorithm, GA), optimización por colonia de hormigas (Ant Colony Optimization, ACO), recocido simulado (Simulated Annealing, SA), optimización por enjambre partículas (Particule Swarm Optimization, PSO), evolución diferencial (Differential Evolution, DE), búsqueda armónica (Harmony Search, HS) y Procedimiento de Búsqueda del Pescador (Fisherman Search Procedure, FSP). Estos algoritmos buscan en el espacio de soluciones bajo ciertas reglas independientes de las características del problema a resolver. A diferencia de los métodos numéricos, la función objetivo no requiere ser diferenciable o continua, por tanto, los algoritmos metaheurísticos pueden ser aplicados para resolver todo tipo de problemas de optimización.

2.2 Estado del Arte

2.2.1 Algoritmos basados en búsqueda armónica

Algunas de las soluciones propuestas para problemas de optimización binaria en el marco de la búsqueda armónica (Harmony Search, HS) consisten en convertir variables de decisión reales a valores discretos; una estrategia comúnmente usada, consiste en reemplazar un número real con su entero más cercano. Frente a este hecho, Zou Gao et al. [13] (2010) desarrollaron un nuevo algoritmo de búsqueda

armónica global (Novel Global Harmony Search, NGHS1) para resolver problemas 0/1 Knapsack tomando conceptos de enjambre de partículas, además, reemplaza la consideración de la memoria armónica y el ajuste de tono con un nuevo esquema de posiciones y estrategia de mutación.

Wang et al. [14] (2013) propusieron un algoritmo de búsqueda adaptativa binaria armónica (Adaptive Binary Harmony Search, ABHS), que incluye una estrategia adaptativa escalable para mejorar la habilidad de búsqueda y robustez del HS original. ABHS fue evaluado con problemas 0/1 Knapsack y sus resultados numéricos demostraron que este algoritmo es más efectivo para resolver problemas de programación binarios. Más adelante se extendió ABHS (AHBS1)[15] (2013) para encontrar el valor óptimo de los parámetros controladores del algoritmo y así mejorar el control sobre su rendimiento. ABHS y ABHS1, demostraron su éxito, pero necesitan afinar demasiados parámetros. Recientemente un algoritmo de búsqueda armónica discreta global (Discrete Global Harmony Search, DGHS)[16] (2014) fue propuesto para resolver problemas 0/1 Knapsack con programación binaria. En esta propuesta se incluye un operador reparador de dos fases para generar las nuevas soluciones, pero su excesivo uso hace que sea fácil estancarse en óptimos locales.

Finalmente en 2015, Kong et al. [1] propusieron un algoritmo para problemas 0/1 Knapsack usando búsqueda binaria simple (Simple Binary Harmony Search, SBHS), con el objetivo de reducir la cantidad de parámetros utilizados y mejorar su rendimiento. El algoritmo propuesto se comparó con 14 variantes de HS, de las cuales 9 son continuos y 5 son binarias. Al finalizar, se concluyó que SBHS tiene mejor habilidad de optimización y escalabilidad. Aun así, ellos admiten que su algoritmo puede mejorarse con variables que se adapten a medida que el proceso avanza.

2.2.2 Algoritmos basados en optimización por enjambre de partículas

Shi. [17] en 2006 propuso un algoritmo basado en colonia de hormigas capaz de escapar de óptimos locales ajustando dinámicamente las hormigas. Luego en 2007, Wang et al.[18] propusieron un algoritmo basado en enjambre cuántico de partículas el cual demostró ser más robusto que PSO usando un mecanismo denominado ángulo cuántico. Después Li y Li [19] en 2009 usaron un algoritmo de enjambre de partículas binario basado en múltiples mutaciones (Multiple Mutation Binary Particule Swarm Optimization, MMBPSO), que es capaz de evitar óptimos locales.

Más recientemente en 2014, Bhattacharjee et al. [4] proponen un algoritmo basado en PSO denominado saltos de rana revueltas (Shuffle Frog Leaping Algorithm, SFLA), el cual imita el comportamiento de ranas buscando comida en un estanque. Los autores muestran que pueden encontrar la respuesta correcta con un pequeño margen de error para los problemas estándar 0/1 Knapsack. Para los problemas más grandes (generados aleatoriamente según unas características de correlación entre las variables y sin una respuesta correcta), es posible que quede atrapado en óptimos locales, como solución a esto abordan la posibilidad de mejorar el proceso de mutación.

En 2015, Moosavian et al. [5] proponen una mejora al algoritmo de liga de fútbol (Soccer League Competition, SLC), un algoritmo basado en enjambres de partículas para resolver el problema 0/1 Knapasck. Previamente este algoritmo había tenido éxito para resolver problemas discretos [20, 21] y decidieron competir con diferentes algoritmos que habían tenido éxito resolviendo el problema 0/1 Knapsack, entre ellos ABHS y NGHS. Sus resultados muestran que al usar dos operadores para buscar el óptimo global hace que la respuesta converja más rápido y sea más confiable.

2.2.3 Otros enfoques de solución

Lin [22] (2008) resolvió el problema sin definir funciones miembro para cada coeficiente del peso, haciéndolos imprecisos y de esta forma volver el problema difuso. Según sus resultados, el algoritmo genético se desempeñó mejor con esta lógica difusa.

En 2009, Zhao et al. [23] propusieron un algoritmo para problemas 0/1 Knapsack usando estrategias voraces (greedy), para abordar instancias más difíciles del problema. Los resultados fueron mejores cuando el algoritmo inicializaba la población con estas estrategias.

Liu y Liu [24] (2009) propusieron un algoritmo basado en esquemas y guías (Schema-Guiding Evolutionary Algorithm, SGEA), con un operador de esquema modificado, ajustaron la distribución de la población, así como también construyeron un esquema de agrupación que guía la evolución de los individuos, mejorando su capacidad de búsqueda local y global.

2.2.4 Búsqueda armónica binaria simplificada (SBHS)

Geem et al. [25] proponen en 2001 el algoritmo HS, un algoritmo que posee un método de búsqueda simple pero eficaz, que está inspirado en el proceso de improvisación musical que hacen los cantantes de jazz. En este proceso se busca encontrar una armonía perfecta desde un punto de vista estético, que es similar al proceso de búsqueda de un óptimo global en optimización, evaluado por una función objetivo. En particular, una armonía musical en HS se ve como un vector de variables y la mejor armonía obtenida es análoga a la solución óptima global.

Kong et al. [1] proponen una modificación a HS denominada SBHS, con el fin de optimizar el algoritmo para eludir sus desventajas en problemas binarios y de esta forma acoplarse mejor a la resolución del problema 0/1 Knapsack. En SBHS sólo 2 parámetros deben ser inicializados, el tamaño de la memoria armónica (Harmony Memory Size, HMS) y la rata (porcentaje) de consideración de la memoria armónica (Harmony Memory Consideration Rate, HMCR). Además, define una nueva regla de improvisación basada en la diferencia entre la mejor armonía y una armonía elegida al azar de la memoria armónica (Harmony Memory, HM) para implementar un ajuste al tono sin ningún parámetro. HMCR se incrementa linealmente con la dimensión del problema para mejorar la habilidad de optimización del algoritmo. Para garantizar la factibilidad de las soluciones, se usa un procedimiento de reparación de dos etapas donde se vuelve factible una solución que no es factible.

2.2.5 Competencia de liga de fútbol (SLC)

Moosavian et al. [5] proponen un algoritmo imitando el comportamiento de las ligas profesionales de fútbol. La población se divide en dos niveles, equipos y jugadores, con los cuales se halla el óptimo global de forma iterativa. En una liga de fútbol, durante una temporada, cada equipo juega contra el resto de equipos dos veces, una como visitante y otra como local, para un total de M x (M-1) veces, donde M es el número de equipos. Los equipos reciben tres puntos por victoria, un punto por empate y cero puntos por derrota. Cada equipo es valorado por el total de puntos y el equipo con más puntos al final de la temporada se considera el campeón.

En SLC cada jugador es un vector solución y durante el curso de la temporada, cada equipo juega con los otros equipos una sola vez y los equipos reciben puntos por reemplazar sus jugadores. Jugadores que son más fuertes, constituyen un equipo más robusto que se encuentra en una posición alta de la tabla de la liga. Más aun, equipos más poderosos tienen posibilidades más altas de ganar sus partidos. Sin embargo, no es posible predecir el ganador hasta que termina una partida.

Así como existe una competencia entre equipos, también hay una competencia interna dentro de cada equipo. Los jugadores compiten unos con otros mejorando su rendimiento, con el fin de llamar la atención del entrenador. Esta competencia interna lleva al crecimiento en calidad y poder del equipo.

En cada equipo, existe un jugador clave, llamado jugador estrella (Star Player, SP). SP tiene el mejor rendimiento entre sus compañeros de equipo. Además, existe un jugador único en cada liga que se denomina, jugador súper estrella (Super Star Player, SSP). SSP es definido como el jugador más impactante de la liga [20, 21]. Por ejemplo, en La Liga Española durante 2014, Cristiano Ronaldo fue el SSP de la liga y el SP del Real Madrid, por otra parte, Lionel Messi fue el SP del Barcelona.

Después de cada partido, los jugadores en el equipo ganador adoptan diferentes estrategias para mejorar su rendimiento futuro. Cuando un equipo gana un partido, los jugadores tratan de imitar al SP del equipo y el SSP de la liga, los operadores de imitación y provocación en SLC imitan esta estrategia. En este estudio se introduce una mejora al operador de imitación basado en el algoritmo de búsqueda armónica (HS) el cual mejora el rendimiento de SLC para encontrar un óptimo global.

2.2.6 Saltos de ranas revueltas (MDSFL)

Bhattacharjee et al. [4] proponen un algoritmo basado en como las ranas buscan comida en un estanque. La población de este algoritmo está compuesta de ranas (soluciones), las cuales se reparten en varios subgrupos denominados memeplexes. Cada memeplex es considerado una cultura diferente de ranas, el cual desempeña una búsqueda local. Cada rana en su memeplex trata de saltar al lugar donde se encuentra la mayor cantidad de comida (mejor solución) basándose en las ranas que ya se encuentran más cerca: A este concepto de acercamiento se le denomina “idea”. Después de un número de iteraciones todas las ranas se revuelven en los diferentes memeplexes con el propósito de que compartan sus “ideas” con las otras

ranas. Este proceso se repite hasta converger en una respuesta o se cumplan ciertas condiciones de parada. El proceso de acercamiento a la respuesta se lleva a cabo cuando la peor rana de un memeplex salta para aproximarse a la mejor rana de dicho memeplex, en caso de que el salto no haya generado una mejor respuesta, se usa a la mejor rana global en lugar de la mejor rana del memeplex, y si aún no genera una mejor respuesta, se genera una rana aleatoriamente.

El algoritmo también fue utilizado para resolver otros problemas relevantes como el diseño de una red de distribución de agua, con resultados favorables [26, 27], el problema de despacho de ferretería [28], problema de planificación para tiendas con alto flujo [29], Jiles-Atherton (JA) modelo de determinación de parámetros [30], problema de programación de proyectos restringido por recursos [31], estimación del modelo de parámetros de un motor doble jaula asíncrono [32], el problema de despacho de carga económica [33], la evaluación de desempeño de distribución de redes radiales [34] y el problema de enrutamiento de vehículos [35].

2.2.7 Procedimiento de Búsqueda del Pescador (FSP)

El Procedimiento de Búsqueda del Pescador (Fisherman Search Procedure) es una metaheurística propuesta recientemente para resolver problemas de optimización global. Esta metaheurística propuesta en [2] (2014) se inspira en el comportamiento y destrezas observadas en un pescador para hallar el mejor lugar de pesca. El procedimiento para realizar la búsqueda inicia por identificar los puntos de captura iniciales que serán las posibles soluciones al problema en el área de pesca (el espacio de búsqueda), se escoge un punto de captura desde el cual se iniciara a lanzar la red de pesca un determinado número de veces, la red abarca un área, en esta los puntos de red ayudan a realizar una búsqueda local que permite generar nuevos puntos de captura, explotando esta área, con los nuevos puntos de captura el pescador evalúa cual es el mejor para realizar los nuevos lanzamientos y repetir el proceso de búsqueda reemplazando el anterior punto de captura. El proceso se

realiza para cada punto de captura encontrado inicialmente, favoreciendo la exploración del área de pesca, sin olvidar los mejores puntos de pesca encontrados en los anteriores lanzamientos. Cuando termina el proceso se elige el mejor punto de pesca encontrado y esa será la solución.

FSP ha sido comparado con otras tres metaheurísticas [2]: procedimiento voraz adaptativo probabilístico (Greedy Randomized Adaptive Search Procedure, GRASP), Evolución Diferencial (Diferential Evolution, DE) y Optimización de Enjambre de Partículas (Particle Swarm Optimization, PSO), obteniendo los mejores resultados para problemas continuos.

2.2.8 Algoritmo genético de pares monógamos (MopGA)

Lim et al. [6] diseñaron un algoritmo específico para enfrentar el problema 0/1 knapsack. Este algoritmo se basa en el comportamiento de sociedades monógamas, donde una pareja se dedica a reproducirse, generando nuevos individuos para la población. Cada pareja se reproduce durante un determinado número de iteraciones, generando nuevos hijos que compiten con los anteriores por un espacio en la población, tomando solamente los mejores hijos por cada iteración. Cada pareja tiene la posibilidad de ser “infiel” y reproducirse con un individuo aleatorio de la población (siendo un individuo distinto a la pareja original).

Este algoritmo está basado en el comportamiento animal de aves y lagartos monógamos, además de la sociedad humana [6]. Este algoritmo comparo su rendimiento y tasa de éxito frente a un algoritmo genético estándar y un algoritmo genético paralelo que usa CUDA [36].

2.2.9 Algoritmo de búsqueda discreta gravitacional

Sajedi et al. [37] se han basado en el comportamiento de las partículas newtonianas para resolver el problema 0/1 knapsack. Se comparan con varios algoritmos entre ellos, el de Competencia de Liga de Fútbol, pero las instancias utilizadas no superan los 1500 objetos y su tasa de éxito no es del 100%.

2.2.10 Grado voraz y expectativa de eficiencia

Lv et al. [38] crearon un algoritmo modular de 3 etapas para resolver el problema 0/1 knapsack. Se comparan con Saltos de Rana Revueltas pero las instancias utilizadas no superan los 200 objetos y su tasa de éxito no es del 100%.

3. PROCEDIMIENTO DE BÚSQUEDA DEL PESCADOR

En este capítulo se describen las dos propuestas desarrolladas para este trabajo de grado para solucionar el problema 0/1 Knapsack para grandes dimensiones. La primera el Procedimiento de Búsqueda del Pescador Binario Secuencial (Sequential Binary Fisherman Search Procedure, SBFSP) y la segunda el Procedimiento de Búsqueda del Pescador Binario paralelo usando la GPU, (Parallel Binary Fisherman Search Procedure using GPU, PBFSPG). Estos algoritmos se inspiraron en FSP y requirió varios cambios, entre ellos, el cambio de la representación de un espacio continuo a uno binario y el cambio de los procedimientos de búsqueda local a métodos que permitieran una mejor solución para las distintas clases de problemas 0/1 Knapsack (sin correlación, débilmente correlacionados y fuertemente correlacionados).

Es preciso comentar que en el desarrollo de este trabajo se realizó una propuesta denominada BFSP [39] (Binary Fisherman Search Procedure), que resuelve instancias de pequeñas y medianas dimensiones del problema 0/1 Knapsack. Los resultados con este algoritmo se publicaron a nivel internacional, el 14 de septiembre de 2016 en el evento MAEB 2016 en Salamanca (España). Aunque los resultados de BFSP fueron prometedores, el algoritmo que aquí se presenta es una mejora que al ser implementado en paralelo obtiene resultados mucho mejores y en tiempos de ejecución reducidos.

3.1 SBFSP para resolver el problema 0/1 knapsack

En la Figura 3-1 se muestra de manera simplificada la evolución de un punto de captura durante t igual a tres iteraciones. En dicha figura, los puntos de captura x_i tienen a su alrededor diversos nudos y_{ij} de la red de pesca, cuando se lanza en cada punto. El algoritmo hace optimización local en cada uno de los puntos de captura,

seleccionado un nudo que sea mejor que el punto de captura y moviéndose allí para repetir el proceso, similar a un algoritmo de ascenso a la colina.

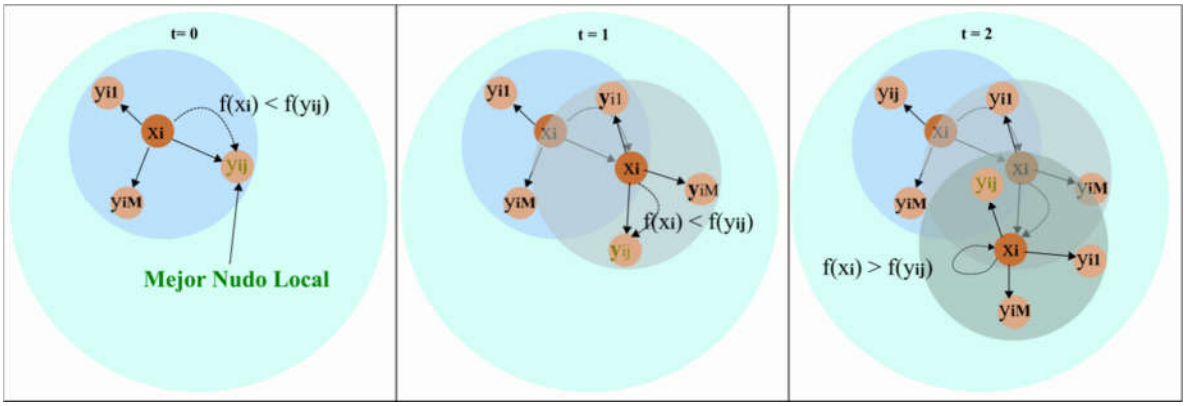

Figura 3-1 Evolución de un punto de captura en SBFSP

La optimización local alrededor de un punto de captura Figura 3-1 se muestra inicialmente en la iteración t = 0, allí el punto de captura x_i genera M nudos viables a su alrededor, cada nudo y_{ij} de la red del punto de captura i se obtiene mutando o usando HUX [40] respecto a x_i y reparando esa mutación. Al procedimiento de generar los nudos se le denomina lanzar la red, este procedimiento es diferente al propuesto en BSFP y FSP, que se basa en búsqueda guiada y búsqueda local, para SBFSP se hace una búsqueda local de ascenso a la colina sobre un vecindario viable y variable generado con diversas estrategias y mucho más intensamente (M recibe valores más grandes que en BFSP y FSP), con esto se eliminó la búsqueda guiada, que involucraba un parámetro adicional L (número de lanzamientos) hecho que además facilita el proceso de afinamiento y resuelve el problema de concurrencia en la versión paralela.

Luego de lanzada la red se busca el mejor nudo y si el valor de la función objetivo del mejor nudo mejora, el punto de captura actual se reemplaza por el mejor nudo como se muestra en la Figura 3-1 para t igual a 0 y 1, si el valor no mejora entonces el punto de captura no se mueve como se muestra en la Figura 3-1 para t igual a 2.

El proceso de optimización que se muestra es para un solo punto de captura, este se debe realizar para los N puntos de captura que se definen aleatoriamente al iniciar el algoritmo, como se muestra en la Figura 3-2 línea 2.

Algoritmo SBFSP(T, N, M, p_m, p_b, iMáx)

```
1  PARA i HASTA N
2     Crear aleatoriamente x_i
3     Reparar x_i usando reparación aleatoria
4     Evaluar x_i
5     c_i ← 2; p_i ← copia(x_i)
6  FIN PARA
7  Actualizar x_g
8  L ← 0        // número de iteraciones que x_g no mejora
9  p_mL ← p_m  // p_m(L)
10 PARA t = 0 HASTA T          // iteraciones
11    PARA i = 0 HASTA N       // puntos de capturas
12       PARA j = 0 HASTA M  // nudos
13          SI U(0,1) ≤ p_mL ENTONCES
14             Crear y_ij usando mutación por HUX
15             Reparar y_ij usando reparación por densidad
16          SINO
17             Crear y_ij usando mutación encendiendo p_b
18             Reparar y_ij usando reparación por valor
19          FIN SI
20          Evaluar y_ij
21          SI fn(p_i) < fn(y_ij) ENTONCES
22             p_i ← copia(y_ij)
23          FIN SI
24       FIN PARA // fin nudos
25    FIN PARA    // fin puntos de captura
26    PARA i = 0 HASTA N
27       SI fn(x_i) < fn(p_i) ENTONCES
28          x_i ← copia(p_i)
29       FIN SI
30    FIN PARA
31    Actualizar x_g
32    SI x_g mejoro ENTONCES
33       SI p_mL > p_m ENTONCES
34          p_mL ← p_mL − 0.05
35       FIN SI
36       L ← 0
37    SINO
38       SI L == iMáx ENTONCES
39          p_mL ← p_m
40          L ← 0
```

```
41          SINO
42              SI pmL ≤ 1 − pm ENTONCES
43                  pmL ← pmL + 0.05
44              FIN SI
45              L ← L + 1
46          FIN SI
47      FIN SI
48      Buscar el punto de captura xs similar a xg
49      Crear aleatoriamente xs
50      Reparar xs usando reparación aleatoria
51      Evaluar xs
52      cs ← 2; ps ← copia(xs)
53      Actualizar xg
54  FIN PARA // fin iteraciones
55  RETORNAR xg
```

Figura 3-2 Algoritmo SBFSP

La Figura 3-2 mostró el algoritmo SBFSP, a continuación se describen los principales elementos.

3.1.1 Parámetros

Para SBFSP se usan 6 parámetros T, N, M, p_m, p_b, iMáx. T es el máximo número de iteraciones (criterio de parada que puede ser remplazado por tiempo máximo de ejecución), N el número de puntos de captura, M el número de nudos por punto de captura, p_m es la probabilidad de seleccionar al mejor punto de captura de la población para generar un nudo y_{ij} del punto de captura i, p_b es el porcentaje de bits que se encienden en y_{ij} sobre el punto de captura i; e iMáx es el número de iteraciones máxima antes de que p_m(L) regrese a su valor inicial, es decir, pm se ajusta en cada iteración pero cuando se cumplen ciertas condiciones se regresa al valor definido por el usuario.

3.1.1.1 Número de puntos de captura (N)

El número de puntos de captura permite definir N diferentes regiones de búsqueda para realizar el proceso de exploración, un punto de captura es una solución candidata la cual se define como se muestra en la Ecuación (3-1):

$$x_i = \{x_{i1}, x_{i2}, \dots, x_{ik}, \dots, x_{in}\} \; donde \; x_{ik} \; \epsilon \{0, 1\}; \; i = 1, 2, \dots, N \; y \; k = 1, 2, \dots, n \quad \textbf{(3-1)}$$

Donde N es el número de puntos de captura y n (minúscula) es el número de objetos disponibles para incluir en la mochila además un punto de captura tiene asociado un coeficiente de anchura (c_i).

El coeficiente de anchura sirve para saber si el punto de captura mejoró. Si el punto mejoró c_i se establece a 1 y se procede a realizar exploración, pero si el punto no mejoró, entonces c_i será 2 y se procede a realizar explotación. Esta interpretación es diferente a la que tiene del coeficiente de anchura en BFSP y FSP. Mientras se realizaron los experimentos con SBFSP se observó que con la nueva interpretación el algoritmo tiene mayor probabilidad de salir de óptimos locales sobre todo en instancias de alta dimensionalidad (n >= 1000).

3.1.1.2 Número de nudos por punto de captura (M)

El número de nudos por punto de captura M se usa en el proceso de exploración y explotación a partir de cada punto de captura. Alrededor del punto de captura i se generan M nudos, estos nudos son las regiones que se explorarán (mutación encendiendo bits y reparación basada en valor) o explotarán (HUX y reparación basada en densidad). Un nudo es una solución candidata igual que el punto de captura y se define como se muestra en la Ecuación (3-2):

$$y_{ij} = \{y_{ij1}, y_{ij2}, \dots, y_{ijk}, \dots, y_{ijn}\}$$
$$donde\ y_{ijk} \epsilon \{0,1\};\ i = 1,2,\dots,N; j = 1,2,\dots,M; k = 1,2,\dots,n \quad \textbf{(3-2)}$$

3.1.1.3 Probabilidad del mejor (p_m), iteraciones máximas (iMáx) y porcentaje de bits (p_b)

El parámetro probabilidad del mejor p_m es la probabilidad de explotar la región del mejor punto de captura, el número de iteraciones máximas iMáx es el número de iteraciones máximas en las cuales el mejor punto de captura no mejora, y el porcentaje de bits p_b es el número de bits Ecuación (3-3) que se encenderán de los bits apagados del punto de captura durante el procedimiento de mutación por encendido de bits. Estos tres parámetros están relacionados con el procedimiento de lanzamiento de la red y la generación de los nudos de esta.

$$Bits\ encendidos\ y_i = \frac{Bits\ apagados\ x_i \times p_b}{c_i} + Bits\ encendidos\ x_i \quad \textbf{(3-3)}$$
$$c_i \epsilon \{1,2\}$$

La probabilidad del mejor como parámetro debe entenderse como la probabilidad de usar HUX y el procedimiento de reparación en dos etapas basado en la densidad, cuando un número aleatorio U(0, 1) cae dentro de esta probabilidad se está explotando la región del mejor punto de captura; cuando no está dentro de esta probabilidad se usa un procedimiento de mutación en el cual se encienden aleatoriamente un número de bits dados por la Ecuación (3-3), es importante detenerse en este punto para explicar que la cantidad de bits a encender serán mucho mayor a los que generalmente se modifican con otros procedimientos de mutación, y esto se hace por que se usara un procedimiento de reparación de dos etapas basado en valor (valor de cada objeto), si no se hace esta mutación al reparar las soluciones generadas serán similares y realizará explotación en lugar de exploración.

Los parámetros probabilidad del mejor p_m e iteraciones máximas iMáx están relacionados, pm está en función del número de iteraciones en las cuales no mejora el mejor punto de captura, como se muestra en la Ecuación (3-4).

$$p_m(0) \leq p_m(L) \leq p_m(iMax);\ p_m(0) = p_m;\ p_m(iMax) = 1 \quad p_m \qquad \textbf{(3-4)}$$

El objetivo de mover este parámetro entre p_m y 1 - p_m es definir qué tipo de mutación y reparación se realizará a los nudos, dependiendo del valor de $p_m(L)$ dominará la mutación por HUX y reparación por densidad o mutación de encender bits y reparación por valor.

3.1.2 Procedimientos de reparación

Cuando se tiene un problema de optimización con restricciones como el problema 0/1 Knapsack existen dos aproximaciones una basada en penalización y otra basada en reparación [4]; SBFSP es una metaheurística basada en reparación, una de las principales características es que usa tres (3) procedimientos de reparación (simple, basado en densidad y basado en valor) a diferencia de BFSP [39], MDSFL [4] y SBHS [1] que usan sólo uno (1) el basado en densidad [1].

3.1.2.1 Procedimiento de reparación simple

El procedimiento de reparación simple, se usa para reparar exclusivamente los N puntos de captura x_i y generar regiones para la exploración lo más distintas posibles apagando bits aleatoriamente hasta que la solución deje de exceder la capacidad de la mochila.

3.1.2.2 Procedimientos de reparación basados en densidad y valor

Estos procedimientos se usan exclusivamente para reparar los nudos y_{ij} generados durante el lanzamiento de la red, lo que en este trabajo se renombro por simplicidad "procedimiento de reparación basado en densidad". La densidad para un objeto se define como se muestra en la Ecuación (3-5), fue propuesto en [1] con el nombre de "procedimiento de reparación de dos etapas para reparar soluciones inviables", este procedimiento se adaptó para usar valor en lugar de densidad como criterio para seleccionar los mejores objetos.

$$u_k = \frac{v_k}{w_k} \quad \textbf{(3-5)}$$

Como se mencionó en el párrafo anterior los procedimientos se basan en el procedimiento de reparación de dos etapas para reparar soluciones inviables [1]. La primera etapa consiste en convertir en factible una solución no factible removiendo los objetos de menor densidad (durante la experimentación se probó remover los ítems de menor valor y aleatoriamente, los resultados fueron peores a los de usar densidad, debido a que la densidad provee información acerca del tipo de instancia sobre la cual se está trabajando).

La segunda etapa consiste en ingresar objetos a la mochila. Cuando se remueven objetos de la mochila (primera etapa) para no exceder la capacidad de esta, algunas veces la mochila quedara con espacio; en este escenario se ingresan objetos con una de las siguientes estrategias. Se ingresan los ítems de mayor densidad (procedimiento de reparación basado en densidad), o los ítems de mayor valor (procedimiento de reparación basado en valor), mientras no se exceda la capacidad.

El procedimiento para reparar los nudos y_{ij} se muestra en la Figura 3-3, los parámetros del algoritmo son: x la solución inviable, n la dimensión del problema, w

los pesos de los objetos, $w_{máx}$ la capacidad de la mochila, S_w es la secuencia de objetos en orden ascendente por peso, S_u es la secuencia de objetos en orden ascendente por densidad, S_x es la secuencia en orden ascendente de objetos por la cual se va reparar que puede ser densidad (S_u) o valor (S_v).

Antes de empezar el proceso de reparación se debe definir si la solución candidata es inviable, como se muestra en la Figura 3-3 líneas 1 y 5, w_t es la capacidad a almacenar en la mochila, w_c es una variable para decidir si la solución excede la capacidad o no.

Cuando la solución es inviable, se procede a realizar la primera de las dos etapas del procedimiento de reparación, Figura 3-3 líneas 6 a 11, esta etapa consiste en remover los objetos de menor densidad, hasta que la capacidad de la mochila no se exceda. Para calcular si existe espacio aún en la mochila después de esta etapa se buscan si hay objetos que aún pueden adicionarse, Figura 3-3 líneas 12 a 15, es decir aquellos cuyo peso es menor o igual a w_l.

La segunda etapa consiste en ingresar los objetos que todavía pueden adicionarse, el primer paso consiste en ordenar ascendentemente la secuencia S_x' en base a la secuencia S_x y S_w', Figura 3-3 líneas 18 a 29; para finalmente ingresar los objetos en la mochila, Figura 3-3 líneas 30 a 35, hasta que el espacio disponible en la mochila se agote.

```
Algoritmo reparación (x, n, w, w_máx, S_w, S_u, S_x)
1  w_t ← 0
2  PARA k ← 1 HASTA n HACER
3     w_t ← w_t + x[k] × w[k]
4  FIN PARA
5  w_c ← w_t − w_máx; k ← 1
6  MIENTRAS (w_c > 0) Y (k ≤ n) HACER
7     SI x[Su[k]] = 1 ENTONCES
8        x[Su[k]] ← 0; w_c ← w_c - w[S_u[k]]
9     FIN SIN
10    k ← k + 1
11 FIN MIENTRAS
12 w_L ← −w_c; k ← 1
13 MIENTRAS (k ≤ n) Y (w_L ≥ w[S_w[k]]) HACER
14    k ← k + 1
15 FIN MIENTRAS
16 k ← k − 1
17 SI k > 0 ENTONCES
18    PARA i ← 0 HASTA k HACER
19       S_w'[i] ← S_w[i]; S_x'[i] ← i
20    FIN PARA
21    PARA i ← 1 HASTA k HACER
22       PARA j ← i + 1 HASTA k HACER
23          SI S_x[S_w'[S_x'[i]]] < S_x[S_w'[S_x'[j]]] ENTONCES
24             aux ← S_x'[i]
25             S_x'[i] ← S_x'[j]
26             S_x'[j] ← aux
27          FIN SI
28       FIN PARA
29    FIN PARA
30    MIENTRAS (k > 0) Y (w_L > 0) HACER
31       SI (x[S_w'[S_x'[k]]] = 0) Y (w_L ≥ w[S_w'[S_x'[k]]])
ENTONCES
32          x[S_w'[S_x'[k]]] ← 1; w_L ← w_L - w[S_w'[S_x'[k]]]
33       FIN SI
34       k ← k − 1
35    FIN MIENTRAS
36 FIN SI
```

Figura 3-3 Procedimiento de reparación basado en S_x

Para la implementación paralela usando la GPU del procedimiento de reparación las líneas 18 a 35 de la Figura 3-3 tuvieron un tratamiento especial y se reemplazan por el seudocódigo de la Figura 3-4 este seudocódigo realiza el mismo proceso.

```
SI k > 0 ENTONCES
   i ← n
   MIENTRAS (i > 0) Y (w_L > 0) HACER
      SI (x[S_x[i]] = 0) Y (w_L ≥ w[S_x[i]])
ENTONCES
         x[S_x[i]] ← 1
         w_L ← w_L − w[S_x[i]]
      FIN SI
      i ← i − 1
   FIN MIENTRAS
FIN SI
```

Figura 3-4 Etapa dos GPU

3.1.3 Mutación

El conjunto de nudos de tamaño M, generados a partir de un punto de captura, durante el lanzamiento de la red, representan una red de pesca para ese punto de captura. Los nudos para un punto de captura se crean haciendo una mutación sobre el punto de captura. Usando el parámetro p_m se define qué proceso de mutación se usará. Se genera un número aleatorio U(0, 1) si el numero está dentro de la probabilidad se usa el procesamiento de mutación usando HUX, sino se usa el procedimiento de mutación de encender bits.

3.1.3.1 Cruce uniforme medio (HUX)

El cruce uniforme medio se usa para acercarse a la mejor solución. El cruce se realiza entre el mejor punto de captura y el punto de captura actual, el procedimiento para crear un nudo es el siguiente: Se calcula la distancia de Hamming y aleatoriamente se seleccionan la mitad los bits que son diferentes del mejor punto de captura y se le asignan al punto de captura actual.

3.1.3.2 Encendiendo bits

A partir de un punto de captura se creará un nuevo nudo, para esto se van a encender el número de bits dados por la Ecuación (3-6), donde se calcula el número de bits apagados del punto de captura actual se multiplica por el parámetro porcentaje de bits p_b y se divide entre el coeficiente de anchura de punto de captura actual c_i.

$$\frac{Bits\ apagados\ x_i \times p_b}{c_i}, c_i \epsilon \{1, 2\} \quad \textbf{(3-6)}$$

3.1.4 Diversidad

El procedimiento de diversidad líneas 48 a 52 de la Figura 3-2 permite explorar nuevas regiones de búsqueda y escapar de óptimos locales, este procedimiento se aplica al punto de captura x_i similar al mejor punto de captura x_g es decir, el punto de captura x_s con menor distancia de Hamming.

Para aumentar la oportunidad de llevar el punto de captura a nuevas regiones de búsqueda el punto de captura se genera aleatoriamente, se lo repara removiendo objetos aleatoriamente, y el coeficiente de anchura se lo fija en 2.

3.2 Computación de propósito general sobre la GPU

La computación de propósito general sobre la GPU es usada para problemas que expresan algún grado de paralelización en el cómputo de los datos es decir las operaciones pueden realizarse independientemente en hilos (threads) paralelos. CUDA es una plataforma de computación de propósito general sobre la GPU y un modelo de programación, introducida en el noviembre del año 2006 por Nvidia, para

resolver gran variedad de problemas complejos de manera más eficiente que una CPU [41].

Como plataforma CUDA soporta varios lenguajes como C/C++, Java, Python, Fortran, para este trabajo de grado se usó CUDA C/C++ una extensión del leguaje C/C++, para programar en la GPU; y está disponible para las arquitecturas de GPU NVIDIA Pascal, Maxwell, Kepler y Fermi.

Como modelo de programación es escalable entre referencias y arquitecturas para las diferentes GPUs que soportan CUDA, este es un aspecto importante a tener en cuenta ya que el paralelismo continua creciendo con la Ley de Moore [41].

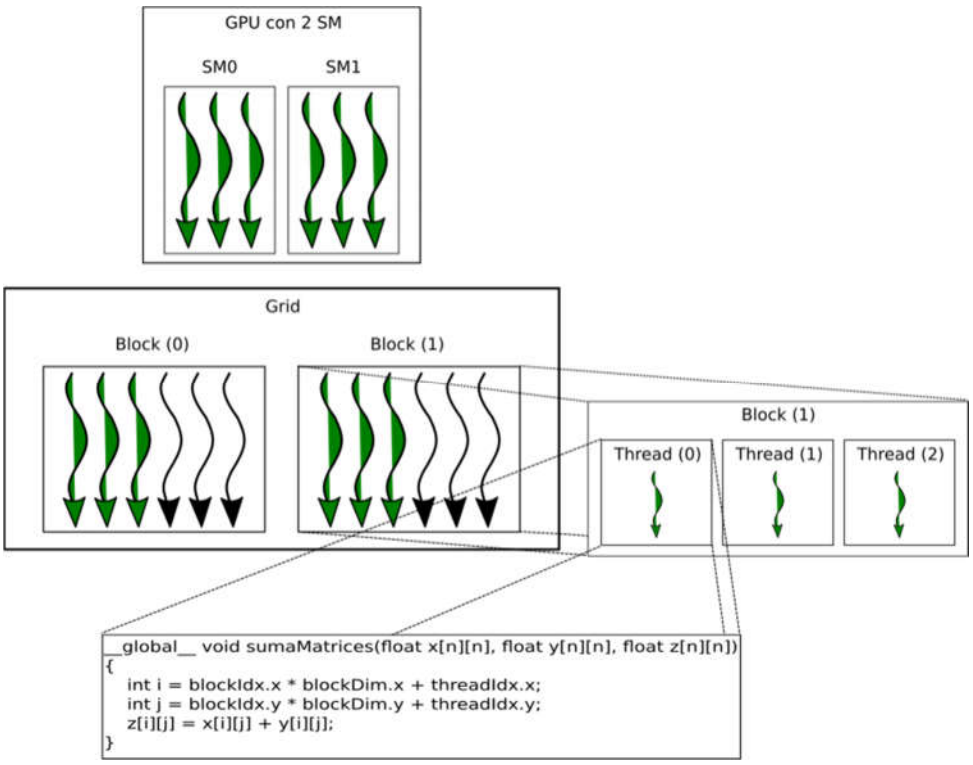

Figura 3-5 Ejecución del kernel sumaMatrices usando CUDA C/C++

3.2.1 Kernels, grids, blocks y threads

La funciones que se ejecutan en la GPU o dispositivo se denominan kernels Figura 3-5. Cuando se ejecuta un kernel o se lanza el kernel; N CUDA threads ejecutan en paralelo las instrucciones del kernel. Es importante tener en cuenta que no todo proceso es paralelizable, cuando esto sucede existen instrucciones que se ejecutan en la CPU o host y otros en el dispositivo, a esto se le denomina programación heterogénea donde se asume que el host y dispositivo administran su recurso de memoria independientemente. Cuando el host y el dispositivo se comunican para intercambiar información de las memorias se conoce como computación hibrida CPU-GPU y es uno de los principales cuellos de botella para el tiempo de ejecución, por esta razón su uso debe ser el mínimo posible [42].

Los kernels en CUDA C/C++ se define con el macro *__global__ sumaMatrices (parámetros)* Figura 3-5 y se lanza usando la siguiente sintaxis <<<*número de blocks por grid, número de threads por block*>>>*kernel-x (parámetros)* en la Figura 3-5 <<<4, 6>>>sumaMatrices (parámetros). En CUDA C/C++ cuando se lanza un kernel se crea una grid que es un conjunto de blocks, un block o block de threads es un conjunto de CUDA threads y tiene un índice dado por la estructura dim3 que es estructura C que almacena las posiciones “x”, “y” y “z” del bloque dentro de la grid. Un CUDA thread ejecuta las instrucciones de un kernel y tiene un índice dentro del block dado por la estructura dim3.

3.2.2 Streaming Multiprocessors y warps

Una GPU Nvidia a razón del modelo y arquitectura tiene un numero de bloques por grid máximo, al igual que un número de threads por block, a nivel hardware tiene un conjunto limitado de SM (Streaming Multiprocessors) y cada SM tiene un conjunto de threads físicos o warp (conjunto de 32 threads), al ser lanzado un kernel la GPU prepara un cronograma de ejecución de los threads en el hardware de esta. Un

bloque es asignado a un SM y el primer warp es enviado a ejecutarse, hasta que todos los bloques y sus threads hayan sido ejecutados. Dicho lo anterior es importante aclarar que el número de blocks por grid y número de threads por block, definidos entre <<< y >>>; si superan las limitaciones a nivel hardware no se ejecutaran exactamente en paralelo, sino que habrá un grupo que se ejecutara en paralelo y luego otro hasta ejecutar todos los threads, por esta razón es importante balancear el número de bloques y el número de threads para aprovechar al máximo la paralelización usando GPUs; por ejemplo en la Figura 3-5, se observa un caso hipotético de una GPU con 2 SM y un warp de 3 threads (recuerde que los warp son un conjunto de 32 threads no existen a nivel físico un número menor) si se tiene un kernel *<<<1, 24>>>sumaMatrices*, en este caso solo habrá un block que será asignado a un SM y que hará un cronograma para los 24 threads, lo cual repercute en el tiempo de ejecución; la mejor forma de hacerlo sería *<<<8, 3>>>sumaMatriz* 2 bloques son asignados a 2 SM y los 3 threads pasan a ejecutarse directamente en un uso correcto de CUDA y aprovechando al máximo la computación de uso general sobre la GPU, sin embargo como se muestra en la Figura 3-5 a veces no es posible ya que un bloque puede representar una unidad conceptual como nudos de una red o memeplexes, en estos casos se debe hacer un correcto ajuste de parámetros del algoritmo para que use al máximo los recursos físicos.

También hay que tener en cuenta que no todos los procesos tienen el mismo nivel de paralelización, en este trabajo se han identificado cuatro (4) tipos de procesos:

- Secuenciales ejecutados en la CPU,
- Secuenciales ejecutados en la GPU -kernel lanzado con un block un thread-
- Paralelos ejecutados en la GPU -kernel lanzado con un block múltiples threads-; y
- Paralelos ejecutados en la GPU -kernel lanzado con múltiples blocks con múltiples threads por block-.

3.3 PBFSPG para resolver el problema 0/1 knapsack

PBFSPG es la metaheurística base para SBFSP; los conceptos explicados en la sección anterior se aplican en esta.

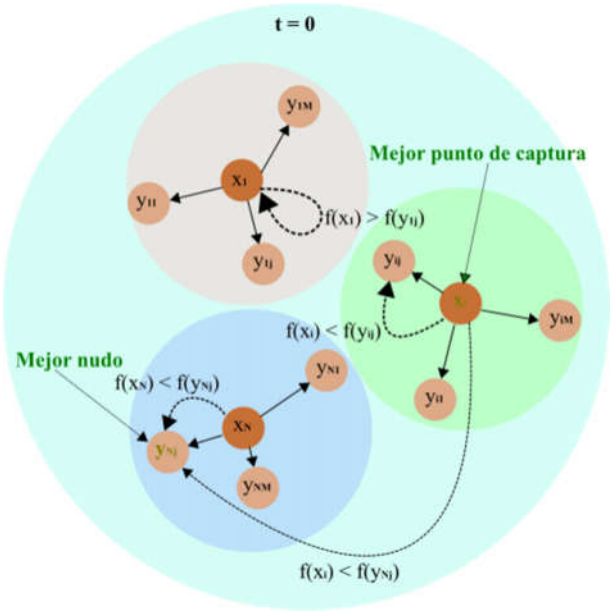

Figura 3-6 Evolución de los puntos de captura en PBFSPG

La Figura 3-6 muestra la iteración (t = 0) de PBFSPG; el algoritmo comienza con la creación en paralelo de los puntos de captura x_i, es decir se están ejecutando N threads a la vez. A partir de los puntos de captura generados y en paralelo se lanza la red o el conjunto de nudos y_{ij} para el punto de captura x_i, en este caso se ejecutan M por N threads a la vez. Luego de lanzada la red se busca cual fue el mejor nudo de esta y sólo si el mejor nudo es mejor que el punto de captura x_i se reemplaza, el proceso de lanzar la red y reemplazar el punto de captura x_i se realiza en paralelo y se ejecutan N threads. La iteración termina con un procedimiento secuencial de buscar el mejor punto de captura, si el mejor punto de captura es el valor óptimo entonces el algoritmo termina, sino se hace una nueva iteración hasta que el algoritmo encuentre el óptimo o termina por el criterio de parada definido (iteraciones o tiempo de ejecución).

3.3.1 Parámetros

PBFSPG usa los mismos seis (6) parámetros que SBFSP; estos están descritos en la sección 3.1.1 Parámetros.

3.3.2 Procedimiento de reparación

Al igual que SBFSP, PBFSPG usa tres procedimientos de reparación simple, basado en densidad y basado en valor descritos en la sección 3.1.2 Procedimientos de reparación.

3.3.3 Algoritmo paralelo usando la GPU

El algoritmo de PBFSPG se muestra en la Figura 3-7, a continuación se describen los kernels usados en la implementación.

3.3.3.1 Inicialización de los puntos de captura

La inicialización se muestra entre las líneas 2 a 5 de la Figura 3-2. Este kernel se lanza usando un block con N threads.

3.3.3.2 Actualización del mejor punto de captura

La actualización se realiza buscando entre todos los puntos de captura aquel con mayor valor de evaluación de la función objetivo, este es un proceso secuencial sin embargo para reducir el uso de computación híbrida CPU-GPU se lanza el kernel usando un block con un thread.

```
Algoritmo PBFSPG(T, N, M, pm, pb, iMáx)
1   Inicialización de los puntos de captura
2   Actualización del mejor punto de captura
3   L ← 0        // número de iteraciones que xg no mejora
4   pmL ← pm // pm(L)
5   PARA t ← 1 HASTA T HACER
6     Lanzamiento de la red de pesca
7     Actualizar los puntos de captura
8     Actualización del mejor punto de captura
9     SI xg mejoro ENTONCES
10      SI pmL > pm ENTONCES
11        pmL ← pmL – 0.05
12      FIN SI
13      L ← 0
14    SINO
15      SI L == iMáx ENTONCES
16        pmL ← pm
17        L ← 0
18      SINO
19        SI pmL ≤ 1 – pm ENTONCES
20          pmL ← pmL + 0.05
21        FIN SI
22        L ← L + 1
23      FIN SI
24    FIN SI
25    Buscar el punto de captura xs similar a xg
26    xs ← nuevo punto de captura generado aleatoriamente
27    Actualización del mejor punto de captura
28  FIN PARA // fin iteraciones
29  RETORNAR xg
```

Figura 3-7 Algoritmo PBFSPG

3.3.3.3 Lanzamiento de la red de pesca

El lanzamiento de la red de pesca es el proceso de crear los nudos como se muestra en la Figura 3-2 entre las líneas 13 a 20. Este kernel se lanza usando N blocks con M threads por block.

3.3.3.4 Actualizar los puntos de captura

Para actualizar los puntos de captura se busca el mejor nudo y luego si este es mejor que el punto de captura actual se reemplaza por el mejor nudo en el caso contrario no se reemplaza; este kernel se lanza usando un block con N threads.

3.3.3.5 Buscar el punto de captura xs similar a xg

Este kernel se lanza usando un block con un thread, el proceso que realiza es buscar el punto de captura x_s con menor distancia de Hamming al mejor punto de captura x_g.

3.3.3 Versiones paralelas

PBFSPG es el algoritmo metaheurístico paralelo usando CUDA para solucionar problemas 0/1 knapsack basado en el Procedimiento de Búsqueda del Pescador. Antes de comenzar con la paralelización el primer paso fue obtener una versión secuencial que se considerara tuviera un buen rendimiento que fue BFSP [39] .

La primera versión paralela sólo resolvió instancias pequeñas y medianas con una tasa de éxito del 100%; para las grandes, el primer error que se identifico fue el nivel de paralelización del algoritmo era muy bajo. En el modelo de programación de CUDA se tienen grids, blocks y threads, la primera propuesta (que se denominó BFSP paralelo con un solo bloque) usaba 1 solo block con N threads un thread para cada punto de captura. Para incrementar el uso de los recursos se creó la versión de BFSP paralelo con múltiples bloques, esta versión paralela elimino el concepto de lanzamiento para poder aprovechar al máximo los recursos, además esta versión mejoró los resultados para algunas instancias grandes (n >= 1000).

BFSP paralelo con múltiples bloques es la mejor versión de BFSP a nivel del modelo de programación de CUDA el cual esta descrito en [41], pero a nivel de la calidad de los resultados se tuvo que empezar a modificar BFSP. En esta etapa de la creación de un BFSP paralelo se identificaron que había instancias que eran más difíciles de resolver, estas eran las débilmente correlacionadas, el proceso para llegar a la respuesta fue obtener una tasa de éxito para la instancia con menor dimensión de las instancias grandes, e ir aumentando la dimensión. Cada incremento en la dimensión enseñaba algo nuevo, esta es la razón de la complejidad del algoritmo comparado con las otras metaheurísticas, pero que le da la capacidad para resolver instancias pequeñas, medianas y de grandes dimensiones.

Entre lo que se aprendió se encuentra: la dimensión n = 500 evidenció que un solo procedimiento de reparación no basta para llegar a la respuesta; como se estaba estancando en un óptimo, se decidió crear un procedimiento de reparación basado en valor, esto funcionó para las instancias n = 1000. En n = 2000 se quedaba en un óptimo local muy lejano al real, se necesitaba más exploración por eso se adicionó el procedimiento de reparación que remueve aleatoriamente objetos en la inicialización. En n = 5000 el problema que se identificó fue que se explotaba poco con mutación por HUX el mejor punto de captura, pero no se podía aumentar estáticamente el valor del parámetro p_m que define cuándo se usa mutación por HUX porque eso generaría sobreexplotación de una región y la posibilidad de caer en óptimos locales, entonces se decidió que p_m estuviera en función de las iteraciones $p_m(L)$ y la evolución de mejor punto de captura. Finalmente, para las instancias de n = 10000 se estaba quedando en un óptimo local. Se identificó que las soluciones eran muy similares, por lo tanto, se implementó la diversidad para que el punto de captura similar al mejor empezara todo el proceso evolutivo desde cero.

El proceso anteriormente descrito termino con un BSFP paralelo que resolvía todas las instancias excepto la instancia (10000 débilmente correlacionada) diez mil objetos débilmente correlacionados, esta es la instancia más difícil de resolver. El tiempo

para encontrar el óptimo en la versión final PBFSPG es en promedio 7 minutos, con un tiempo de hasta 20 minutos en el peor de los casos. Para lograr que la tasa de éxito fuese 100% se hizo una mutación mucho más agresiva, para este caso se adicionó el parámetro p_b. La adición no estropeó el trabajo ya realizado pero sí permitió que el algoritmo resolviera todas las instancias de prueba con una tasa de éxito de 100% pero teniendo en cuenta que la instancia de 10000 débilmente correlacionada es excepcionalmente difícil. Después de todo ese proceso, la versión final de la metaheurística paralela basada en el procedimiento de búsqueda del pescador se denominó PBFSPG.

4. METAHURÍSTICAS PARALELAS USANDO LA GPU PARA RESOLVER EL PROBLEMA 0/1 KNAPSACK

En este capítulo se describe la implementación de cuatro metaheurísticas paralelas usando CUDA para resolver el problema 0/1 Knapsack, en la Figura 4-1 se muestra en gris las nuevas metaheurísticas paralelas usando la GPU implementadas para este trabajo de grado que se basan en las soluciones secuenciales, MDSFL [4], MopGA [6], SBHS [1] y SLC [5], seleccionadas del estado del arte.

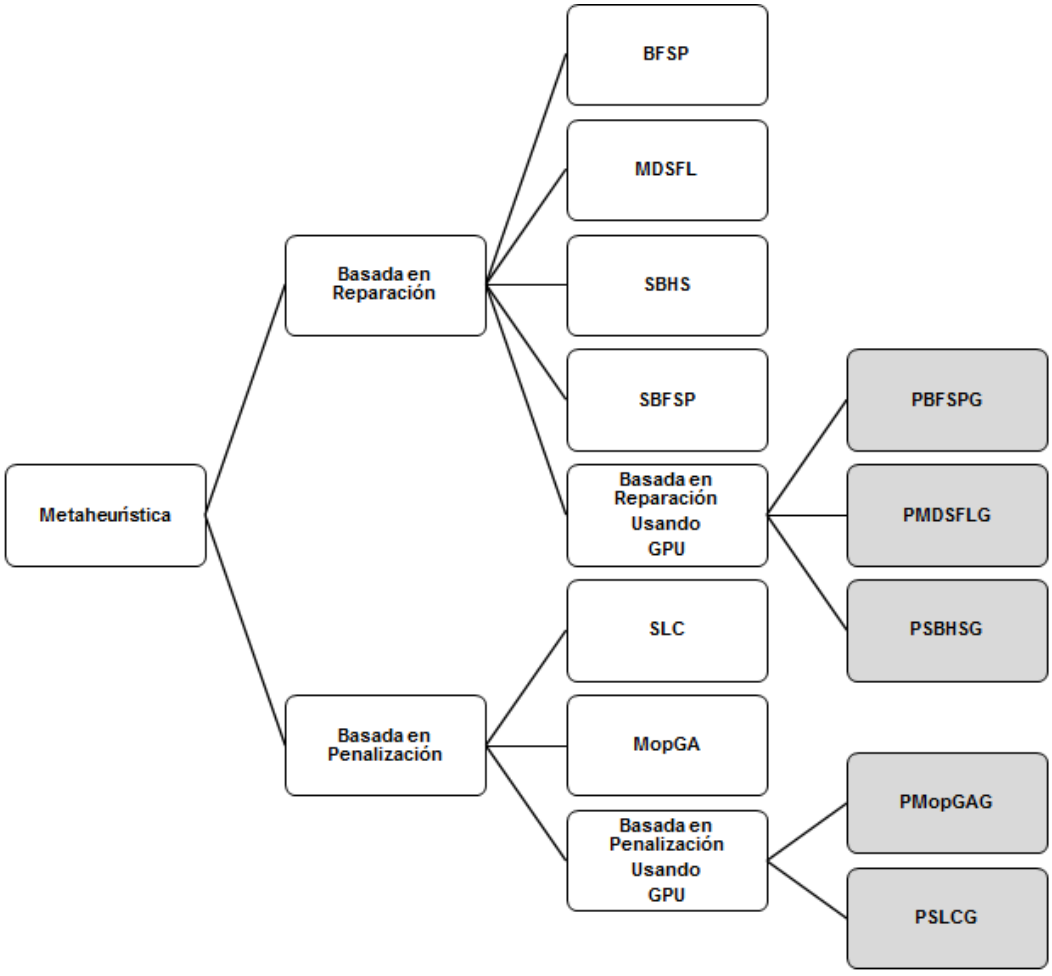

Figura 4-1 Metaheurísticas para el problema 0/1 Knapsack

Antes de comenzar este capítulo se recomienda la lectura del Capítulo 2 donde se describe brevemente cómo funcionan las metaheurísticas MDSFL [4], MopGA [6], SBHS [1] y SLC [5] y la sección 3.2 Computación de propósito general sobre la GPU que explica brevemente el modelo de programación de CUDA.

4.1 Enfoques de solución para las metaheurísticas que solucionan el problema 0/1 Knapsack

El problema 0/1 Knapsack es un problema de optimización discreto binario con restricciones. Cuando se busca este problema en el estado del arte se encuentran dos enfoques para su solución [4] Figura 4-1, el primero usando una función objetivo que penaliza la soluciones candidatas inviables es decir que están por fuera del espacio de búsqueda factible y el segundo realizando una transformación a la solución candidata inviable para convertirla en viable o que esté dentro del espacio de búsqueda factible del problema. A continuación, se describen los dos enfoques.

4.1.2 Soluciones basadas en penalización

Cuando se decide realizar una solución para un problema de optimización con restricciones basado en penalización, el trabajo más arduo está en definir cómo serán penalizadas aquellas soluciones inviables. Para el problema 0/1 knapsack generalmente se usa la Ecuación (2-1) adicionándole un término a la derecha de la ecuación que permite definir cómo será castigada la solución, ejemplos de esto se pueden ver en MopGA [6] y SLC [5] Figura 4-1.

4.1.3 Soluciones basadas en reparación

En una solución que se basa en reparación la función objetivo ya está definida para el problema 0/1 Knapsack como se muestra en la Ecuación (2-1), ahora el esfuerzo se traslada en definir el procedimiento de reparación para que las soluciones no factibles estén siempre dentro del espacio de búsqueda factible, ejemplos de procedimientos de reparación para el problema 0/1 knapsack se pueden ver en la sección 3.3.2 Procedimiento de reparación y en la Figura 4-1 se muestran algunas metaheurísticas que siguen este enfoque.

4.2 Saltos de ranas revueltas usando la GPU (PMDSFLG)

Saltos de ranas revueltas usando la GPU (parallel modified discrete shuffled frog leaping using GPU, PMDSFG) es una metaheurística basada en reparación. La metaheurística base es MDSFL [4] descrita brevemente en la sección 2.2.6 Saltos de ranas revueltas (MDSFL).

4.2.1 Parámetros

PMDSFLG, usa 6 parámetros: el tamaño de la población, P; el número de memeplexes, m; el número de iteraciones para la búsqueda local, it; la probabilidad estática, α, usada en el proceso de variables discretas; la probabilidad de mutación, pm; y el número de nuevas ranas por memeplex, N.

4.2.2 Procedimiento de reparación

El procedimiento de reparación se encuentra descrito en la sección 3.1.2.2 Procedimientos de reparación basados en densidad y valor; pero a diferencia de PBFSPG que usa tres procedimientos de reparación, PMDSFLG sólo usa el

procedimiento de reparación basado en densidad, en este caso el parámetro S_x que recibe el algoritmo de reparación es S_u o la secuencia de objetos ordenados ascendentemente por densidad.

4.2.3 Algoritmo paralelo usando la GPU

La Figura 4-2 muestra el funcionamiento básico del algoritmo paralelo, un memeplex es un conjunto de ranas o soluciones candidatas x_i. A partir de un memeplex se genera un conjunto de nuevas ranas de las cuales se escoge la mejor para reemplazar la peor del memeplex, este reemplazo se da independientemente de si la rana nueva es mejor o no, ya que en la creación de una nueva rana, si esta no es mejor que la peor, genera la nueva rana aleatoriamente para explorar otras regiones del espacio de búsqueda. Es preciso notar que las peores ranas siempre son las últimas dentro de los memeplexes, esto es debido a la forma como se divide la población en dichos memeplexes.

A continuación, se describen los kernels para la solución paralela Figura 4-3

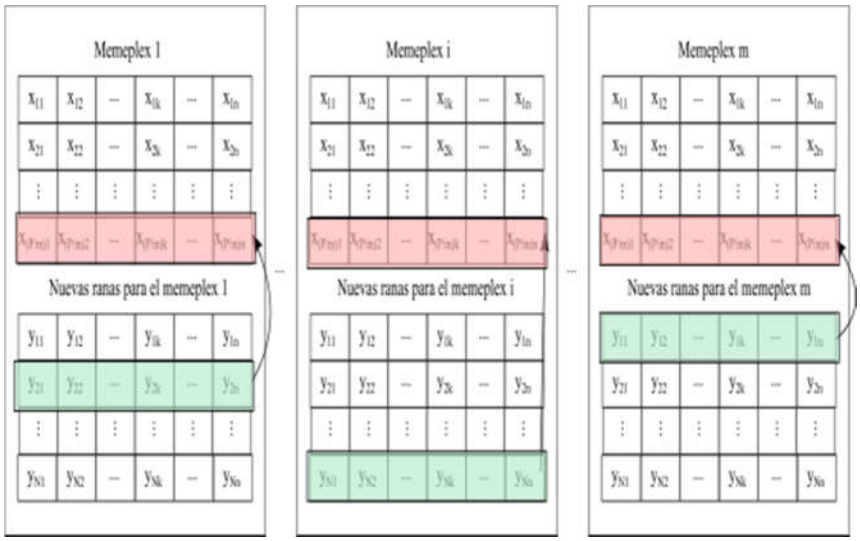

Figura 4-2 MDSFL usando la GPU

4.2.3.1 Inicialización de la población

El número de soluciones candidatas es P, las cuales se generan en paralelo. Para lograrlo el kernel que se lanza tiene m blocks con P/m threads por bloque, cada threads genera aleatoriamente una solución, la reparara usando el procedimiento de reparación basado en densidad y evalúa su aptitud (fitness). Ver línea 1 de la Figura 4-3.

4.2.3.2 División de la población en memeplexes

Un memeplex es un conjunto con un número de P/m ranas, para distribuir la población en cada memeplex el procedimiento es el siguiente, se ordena de mayor a menor la población de ranas según su aptitud, después de esto la mejor rana es la primera, esta se la envía al primer memeplex, la segunda al segundo, la tercera al tercero y así sucesivamente. Se sigue este procedimiento m veces, luego la rana (m + 1) va nuevamente al primer memeplex, la rana (m + 2) al segundo, hasta que las P ranas se hayan dividido en los m memeplexes.

Algoritmo PMDSFLG(P, m, it, α, pm N)

```
1    Inicialización de la población
2    MIENTRAS la condición de parada no se satisfaga HACER
3       División de la población en memeplexes
4       PARA i ← 1 HASTA it HACER
5          Búsqueda local
6          Actualizar la mejor rana global
7       FIN PARA
8       Mutación de la población
9       Actualizar la mejor rana global
10   FIN MIENTRAS
11   RETORNAR la mejor rana global
```

Figura 4-3 Algoritmo paralelo PMDSFLG

En paralelo el procedimiento de división de la población en memeplexes se hace lanzando un kernel con m blocks de P/m threads por cada block. Figura 4-3, línea 3.

4.2.3.3 Búsqueda local

La búsqueda local se usa para que un memeplex evolucione, la creación de una nueva rana sigue el mismo esquema que en MDSFL [4].

En la búsqueda local cuando los memeplexes evolucionan lo hacen de forma secuencial y para generar una nueva rana se necesita saber cuál es la mejor rana global, la mejor rana local, y la peor rana; después de terminada la evolución de un memeplex, el siguiente busca cuál es la mejor rana global, si el memeplex anterior mejoro la mejor rana global, se actualiza la mejor rana. Saber cada vez cual es la mejor rana limita la paralelización del algoritmo, por eso, en la implementación paralela no se hace y los memeplexes hacen su búsqueda local independientemente sin comunicarse con los otros memeplexes para actualizar la mejor rana global.

Al paralelizar se pierde cierto grado de la naturaleza secuencial del algoritmo, para compensar esto, se adicionó el parámetro N (número de nuevas ranas por memeplex). En la versión secuencial sólo se genera 1 nueva rana por memeplex, para el caso de la versión paralela se generan N de las cuales se escoge la mejor para reemplazar la peor rana del memeplex, con esto se intensifica la búsqueda local por cada memeplex.

El kernel encargado de la búsqueda local usa m blocks con N threads en cada block, donde cada thread genera una nueva rana. Luego de generada la rana otro kernel se encarga de buscar la mejor de la N nuevas ranas y reemplazar la peor del memeplex. Este kernel usa 1 block con m threads. Ver línea 5 de la Figura 4-3.

4.2.3.4 Mutación de la población

El parámetro pm le permite decidir a un thread si se muta una rana de la población o no, si se muta el procedimiento de mutación hace lo siguiente; escoge un bit aleatorio entre 1 y n (objetos del problema) y se cambia el valor (bit Flip), se reparará usando el procedimiento de reparación basado en densidad y se evalúa la solución, este kernel usa m blocks con P/m threads por block. Figura 4-3, línea 8.

4.2.3.5 Actualizar la mejor rana global

Finalmente usando un solo kernel con un thread se busca cual es la mejor rana en la población (todos los memeplexes), si no es el óptimo o no se ha cumplido el criterio de parada se repite el proceso desde la división de la población en los memeplexes, Figura 4-3, líneas 5, 9.

4.2.4 Versiones paralelas

Aunque existe una versión paralela de MDSFL [43] no fue usada para la comparación ya que usa múltiples GPUs. Para implementar la versión final para sólo una GPU las implementaciones que se hicieron fueron las siguientes, inicialmente se hizo un algoritmo generando sólo una nueva rana por memeplex y reemplazándola con la peor, de esta forma lo hace MDSFL. Posteriormente con el ánimo de mejorar los resultados para los memeplexes se evaluó la generación de N nuevas ranas (en paralelo) por memeplex reemplazando aquellas nuevas ranas que son mejores que las que hay en el memeplex; esto no funcionó y se decidió regresar al concepto de reemplazar la peor rana del memeplex pero generando N ranas candidatas por memeplex y escogiendo la mejor. Esta última versión obtuvo los mejores resultados.

4.3 Algoritmo genético de pares monógamos usando la GPU (PMopGAG)

El algoritmo genético de pares monógamos usando la GPU (parallel monogamous pairs genetic algorithm using GPU, PMopGAG), es una metaheurística basada en penalización, la base secuencial de esta metaheurística paralela es MopGA [6] descrita brevemente en la sección 2.2.8 Algoritmo genético de pares monógamos (MopGA).

4.3.1 Parámetros

PMopGAG usa cinco parámetros: probabilidad de mutación, p_m; probabilidad de infidelidad, p_i; número de individuos, N (tiene que ser par); el número de iteraciones en que una pareja se mantiene unida, g_p; y el número de nuevos hijos, M.

4.3.2 Función objetivo

MopGA y su implementación paralela PMopGAG se diferencia de las otras metaheurísticas seleccionadas para este trabajo, en que optimiza minimizando Ecuación (4-1). En esta ecuación, λ se conoce como el coeficiente de penalización y es un valor grande [44].

$$\begin{aligned} Min\ F(x) = \quad & f(x) + \lambda \times \max(0, g(x)) \\ & donde\ f(x) = \sum_{k=1}^{n} v_k \times x_k\ ; \\ & g(x) = \sum_{k=1}^{n} w_k \times x_k \quad C; \\ & y\ \lambda = 10^{20} \end{aligned} \tag{4-1}$$

4.3.3 Algoritmo paralelo usando la GPU

En la Figura 4-4 se muestra de manera simplificada el funcionamiento de PMopGAG para la primera iteración; o cuando se forma una nueva pareja usando el complejo de Edipo, en este estado las familias no tienen hijos, el número de familias es N/2. En paralelo las N/2 familias procrean M hijos, de los cuales se escogen los dos mejores para formar parte de la familia y los M – 2 hijos que sobran se descartan.

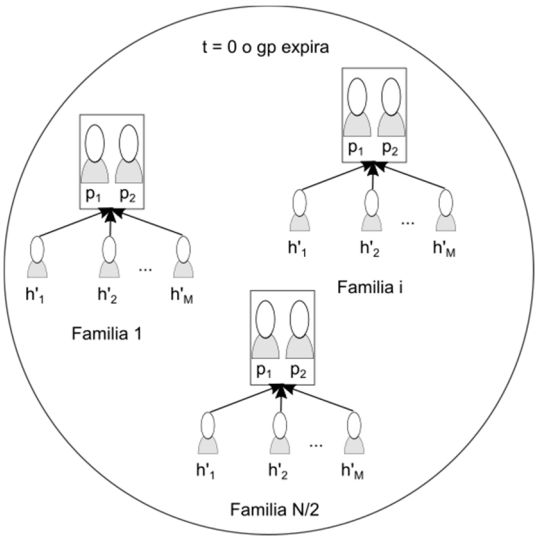

Figura 4-4 Estado de las nuevas familias

La implementación paralela de PMopGAG se describe en las siguientes secciones, el algoritmo paralelo se muestra en la Figura 4-7. Cada sección es un kernel, donde se da una breve descripción de cómo funciona el kernel y cómo se ejecutó.

4.3.3.1 Inicialización de la población

Los individuos se generan aleatoriamente, y se evalúan usando la Ecuación (4-1). Este kernel se lanza usando un block con N threads.

4.3.3.2 Creación de las familias

Aleatoriamente se seleccionan los individuos que generaran una familia, cada familia está compuesta por dos individuos (padres) y el número de familias es N/2. Este proceso se lleva a cabo en la CPU, al terminar se envían las nuevas familias a la GPU.

4.3.3.3 Infidelidad

El proceso de infidelidad se usa para compartir información de otras zonas de búsqueda [6], usando el parámetro probabilidad de infidelidad o Pi se determina qué familia tendrá un caso de infidelidad; el miembro de la pareja que es infiel será el mejor de los padres, y su amante será un miembro aleatorio de las otras familias (no existe incesto en este estado) y puede ser un padre o un hijo de las otras familias. Este kernel se lanza usando un block con N/2 threads.

4.3.3.4 Creación de los hijos

Los hijos se crean, usando cruce uniforme como operador de cruce entre los padres de la familia, y se mutan usando el operador de cambiar un bit (bit Flip), para determinar qué hijo se muta se usa la probabilidad de mutación o pm. Este kernel se lanza usando N/2 blocks con M threads por block. Luego de la primera iteración el estado de las familias se muestra en la Figura 4-5.

4.3.3.5 Reconciliación

Si en la familia hubo infidelidad, esta sólo es temporal, luego de la creación de los hijos las parejas se reconcilian; este kernel se lanza usando un bloque con N/2 threads.

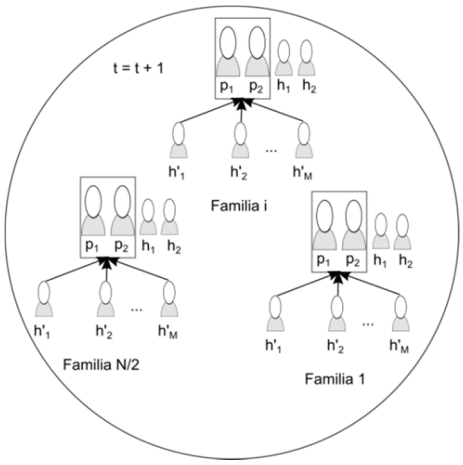

Figura 4-5 Estado de las familias para la iteración t + 1

4.3.3.6 Competencia de la descendencia

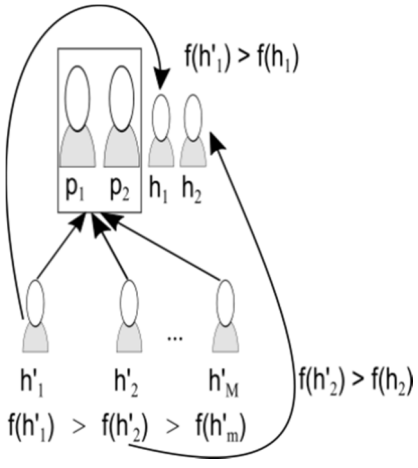

Figura 4-6 Competencia de la descendencia

En la Figura 4-6, se muestra la competencia entre los hijos, esta se realiza entre la nueva generación y la antigua, sólo los mejores hijos pasan a la siguiente generación (aunque en la figura se muestra que el mejor hijo de la nueva generación es el primero no necesariamente es de esta forma). En total se enfrentan (M + 2) hijos de los cuales sólo dos sobreviven. Los otros M hijos son descartados. Este proceso se realiza en la CPU y luego se envía el resultado a la GPU.

4.3.3.7 Sobrevivientes de la competencia de la descendencia

Con el resultado del proceso anterior, en paralelo se reemplazan los hijos de la anterior generación por aquellos que ganaron la competencia; este kernel se lanza usando un block con N/2 threads. Terminado este proceso se busca el mejor de la población compuesta por N individuos y N hijos.

Algoritmo PMopGAG(P_m, P_i, N, g_p, M)

```
1  Inicialización de la población
2  Creación de la población
3  MIENTRAS no se cumpla la condición de terminación HACER
4     SI g_p expira HACER
5        Complejo de Edipo
6     FIN SI
7     Infidelidad
8     Creación de los hijos
9     Reconciliación
10    Competencia de la descendencia
11    Sobrevivientes de la competencia de la descendencia
12 FIN MIENTRAS
13 RETORNAR el mejor de la población
```

Figura 4-7 Algoritmo paralelo PMopGAG

4.3.3.8 Complejo de Edipo

Una familia existe g_p número de iteraciones, luego esta se transforma. Cuando gp expira todas las familias experimentan algo que se denomina complejo de Edipo, donde el mejor padre hace pareja con el mejor hijo, y el peor padre y peor hijo se eliminan, este kernel se lanza usando un block con N/2 threads.

4.3.4 Versiones paralelas

La primera versión paralela se implementó siguiendo la naturaleza de MopGA, es decir, se creaban dos nuevos hijos únicamente. La siguiente versión incrementó el número de nuevos hijos a M, para aumentar la exploración mejorando los resultados. La versión final cambió el concepto de creación de familia cuando gp expira; en MopGA la nueva familia está constituida por el mejor hijo y uno de los hijos de las otras familias, para la versión paralela se usa el complejo de Edipo para crear una nueva familia entregando mejores resultados, ya que en la metaheurística base se estaban perdiendo buenas soluciones, al descartar los mejores padres y sólo dejar los mejores hijos.

4.4 Búsqueda armónica binaria simplificada usando la GPU (PSBHSG)

La búsqueda armónica binaria simplificada usando la GPU (parallel simplified binary harmony search using GPU, PSBHSG) se basa en SBHS [1] descrita brevemente en la sección 2.2.4.

PSBHSG se diferencia de SBHS en que esta propuesta usa múltiples búsquedas armónicas en paralelo (parámetro NHS) cada una con múltiples improvisaciones (parámetro NI) Figura 4-8.

En la Figura 4-8 se muestra de forma simplificada el funcionamiento de PSBHSG donde en paralelo existen diferentes búsquedas armónicas, cada una con su propia memoria armónica la cuales evolucionan independientemente. Cada búsqueda armónica tiene también una memoria para las improvisaciones, las cuales se improvisan en paralelo. Al terminar la iteración t, en paralelo las búsquedas armónicas buscan la peor armonía de la memoria armónica (solución candidata resaltada con rojo) para ser reemplazada por el mejor improviso (solución candidata resaltada con verde), el proceso se repite hasta que se cumpla la condición de terminación.

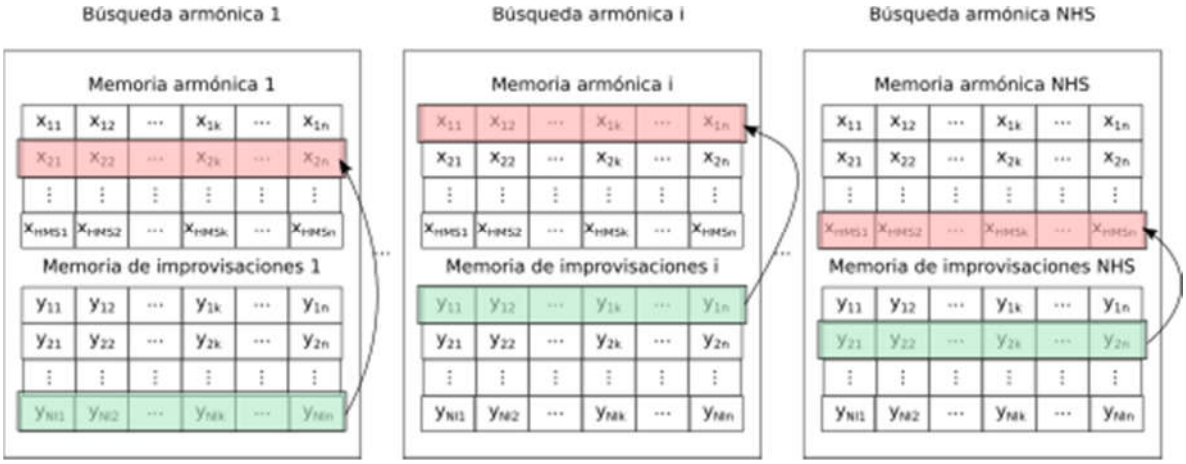

Figura 4-8 SBHS usando la GPU, iteración t

4.4.1 Parámetros

Para PSBHSG se usan 3 parámetros: tamaño de la memoria armónica (harmony memory size, HMS), el número de búsquedas armónicas (number of harmony search, NHS) y el número de improvisaciones (number of improvisations, NI). Aunque existe un parámetro adicional, en la versión original de SBHS, (harmony memory considering rate, HMCR), en la versión paralela no se considera como tal ya que los problemas tienen una dimensión n ≥ 100, y cuando esto ocurre en SBHS el valor de HMCR se actualiza dinámicamente según la Ecuación (4-2) [1].

$$HMCR = 1 \quad \frac{10}{n}, n \geq 100 \qquad \textbf{(4-2)}$$

4.4.2 Procedimiento de reparación

El procedimiento de reparación se encuentra descrito en la sección 3.1.2.2 Procedimientos de reparación basados en densidad y valor en este caso el parámetro S_x que recibe el algoritmo de reparación es S_u o la secuencia de objetos ordenados ascendentemente por densidad.

4.4.3 Algoritmo paralelo usando la GPU

A continuación, se describen los kernels usados en SBHS usando la GPU

4.4.3.1 Inicialización de las memorias armónicas

Una memoria armónica es un conjunto de armonías o soluciones candidatas x_{ij}, i es la búsqueda armónica y j la armonía, como se mostró en la Figura 4-8.

Una armonía es un thread en el cual se genera la solución, se la repara y finalmente se evalúa. Este kernel se lanza usando NHS blocks con HMS threads en cada block.

4.4.3.2 Proceso de improvisación

En la Figura 4-9 se muestra el proceso de improvisación, el esquema ingenioso de improvisación se encuentra descrito en [1].

El kernel para el proceso de improvisación se lanza usando NHS blocks con NI threads por block.

```
Algoritmo improvisación(y_ij)
1  PARA k ← 1 HASTA n HACER
2     SI U(0,1) ≤ HMCR ENTONCES
3        Esquema ingenioso de improvisación para y_ij
4     SI NO
5        SI U(0,1) ≤ 0.5 ENTONCES
6           y_ij[k] = 0
7        SI NO
8           y_ij[k] = 1
9        FIN SI
10    FIN SI
11 FIN PARA
12 Reparar y_ij usando reparación basada en densidad
13 Evaluar y_ij
```

Figura 4-9 Algoritmo de improvisación

4.4.3.3 Actualización de las memorias armónicas

Después de haber improvisado -HMS por NI- armonías, en las NHS búsquedas armónicas se reemplazan las peores armonías, esto se realiza buscando las mejores improvisaciones y reemplazando como se mostró en la Figura 4-8, este reemplazo se realiza si y sólo si, la mejor improvisación para la búsqueda armónica i tiene mejor evaluación de la función objetivo que la peor armonía de la búsqueda armónica i.
Este kernel se lanza usando un block con NHS threads.

4.4.3.4 Actualización de la mejor armonía

El proceso de actualización de la mejor armonía se hace en un kernel con un block de un thread, usado para buscar la mejor armonía entre todas las memorias armónicas. Al terminar el resultado se copia de la memoria de la GPU en la memoria de la CPU.

El seudocódigo de PSBHSG se muestra en la Figura 4-10.

```
Algoritmo PSBHS(HMS, NHS, NI, n)
1  HMCR = 1 – 1 / n
2  Inicialización de las memorias armónicas
3  MIENTRAS no se cumpla la condición de parada HACER
4     Proceso de improvisación
5     Actualización de las memorias armónicas
6     Actualización de la mejor armonía
7  FIN MIENTRAS
8  RETORNAR mejor armonía
```

Figura 4-10 Algoritmo paralelo PSBHSG

4.4.4 Versiones paralelas

SBHS es la única metaheurística de las estudiadas que no es naturalmente paralelizable, ya que su naturaleza es puramente secuencial, aunque en la literatura se encuentran múltiples aproximaciones [45-47].

La primera versión que se implementó de SBHS paralela fue usando múltiples búsquedas armónicas que evolucionan independientemente, pero en cada iteración se copia a todas las búsquedas la mejor armonía global usando un procedimiento de difusión [45]. Luego se añadió un operador que barajaba [46] las armonías y creaba nuevas memorias armónicas. Esta última opción generó muy buenos resultados pero la paralelización realizada no aprovechaba todo el potencial de la GPU. Finalmente se decidió que cada búsqueda armónica creará múltiples improvisaciones en lugar de sólo una (parámetro NI). A las tres versiones anteriores se le añadió la múltiple improvisación y los resultados indicaron que la primera implementación múltiples búsquedas armónicas independientes con múltiples improvisaciones era la mejor versión para el problema.

4.5 Competencia de liga de fútbol usando la GPU (PSLCG)

La competencia de liga de fútbol usando la GPU (Parallel Soccer League Competition using GPU, PSLCG), se basa en SLC [5] que se describe brevemente en la sección 2.2.5 Competencia de liga de fútbol (SLC).

Figura 4-11 Primera ronda (ida) de la liga

La implementación de SLC usando la GPU o PSLCG, puede ser las más familiar para aquellos que conocen como se desarrolla una liga de fútbol. El funcionamiento básico se muestra en la Figura 4-11 la competencia de liga de fútbol es una competencia todos contra todos, para realizar esto un algoritmo define los encuentros con la restricción de que un mismo equipo no puede jugar dos encuentros en la misma jornada, ya que esto generaría un problema de concurrencia dado que los encuentros se realizarán al mismo tiempo en diferentes lugares –este punto podría pensarse trivial, pero no lo es, cuando se realiza una implementación en paralelo muchos threads van a realizar lectura o escritura, si no se tiene en cuenta la concurrencia en la escritura los resultados no serán los esperados. Respecto a la concurrencia de lectura no es un problema siempre y cuando lo que se va leer no

vaya ser modificado. La competencia se desarrolla en jornadas (evitando el problema de concurrencia). Para la primera ronda (ida) después de cada jornada los equipos ganadores aplican a sus jugadores los operadores de imitación y provocación, luego se repiten los encuentros en segunda ronda (vuelta) y se hace el mismo procedimiento. A continuación, se describe más detalladamente los elementos de PSLCG.

4.5.1 Función objetivo

PSLCG es una metaheurística basada en penalización, el valor de la función objetivo se obtiene usando la Ecuación (4-3) [39].

$$\begin{gathered} \text{Max F}(x) = f(x) \quad \lambda \times p(x) \\ donde\ f(x) = \sum_{k=1}^{n} v_k \times x_k\ ; \\ \lambda = \sum_{k=1}^{n} x_k\ ; \\ y\ p(x) = \frac{\sum_{k=1}^{n} w_k \times x_k}{w_{máx}} \quad 1 > 0\ ?\ \frac{\sum_{k=1}^{n} w_k \times x_k}{w_{máx}} : 0 \end{gathered} \tag{4-3}$$

4.5.2 Parámetros

Los parámetros para PSLCG son los mismos que usa SLC, con excepción del parámetro λ o coeficiente de penalización, ya que se calcula como se muestra en la Ecuación (4-3), los demás parámetros son:

El número de temporadas (T) es el número de iteraciones máxima del algoritmo. El número de equipos (nE). El número de jugadores titulares (nT). El número de jugadores sustitutos (nS). HMCR (harmony memory considering rate) y PAR (pitch adjustment rate) los últimos dos parámetros se usan en el operador de imitación.

4.5.3 Algoritmo paralelo usando la GPU

Las secciones que siguen describen los kernels y como fueron lanzados.

4.5.3.1 Inicialización de los jugadores

Este kernel se lanza usando nE blocks cada uno con (Ecuación (4-4)) threads por block; un block representa un equipo y un thread un jugador. El procedimiento que se lleva a cabo dentro del kernel es el siguiente: se genera aleatoriamente un jugador y se evalúa usando la Ecuación (4-3).

$$jugadores\ por\ equipo = nT + nS \quad \textbf{(4-4)}$$

4.5.3.2 Ordenar jugadores

Esta competencia es como cualquier otra, los mejores jugadores estarán en los mejores equipos y los peores en los peores, para realizar esto en PSLCG, se ordenan en orden descendente los jugadores, cada equipo tiene un número total de jugadores (Ecuación (4-4)), el primer equipo toma el número de jugadores que le corresponde, luego el segundo hasta el último equipo.

4.5.3.3 Fijar los jugadores estrella

Cada equipo tiene un jugador que tiene la mejor evaluación, este jugador es el jugador estrella para ese equipó, este kernel lanza un block con nE threads, para buscar cuales son los jugadores estrellas de los equipos.

4.5.3.4 Calcular la potencia del equipo

La potencia de un equipo es la suma de los valores de las evaluaciones de la función objetivo de sus jugadores, el cálculo se hace en paralelo para cada equipo. El kernel se lanza usando un block con nE threads.

4.5.3.5 Realizar jornada

Como se muestra en la Figura 4-11 existen tres jornadas para la primera ronda (ida) para 4 equipos y durante cada jornada se realizarán 2 encuentros. Un algoritmo genera todos los posibles encuentros para la primera ronda, la segunda ronda (vuelta) es simplemente repetir los encuentros; de esta forma la competencia se realiza con todas las combinaciones posibles de enfrentamientos, con la restricción de que un mismo equipo no puede jugar más de una vez por jornada.

Dentro del kernel el proceso para encontrar el ganador del encuentro es por ejemplo para la primera jornada; en el Encuentro 1 entre el equipo E2 contra el equipo E1. El ganador Figura 4-12 se define usando la probabilidad de victoria del E1 Ecuación (4-5).

$$probabilidad\ E1 = \frac{potencia\ E1}{potencia\ E1 + \ potencia\ E2} \quad \textbf{(4-5)}$$

Algoritmo Encuentro(E1, E2)

```
1  SI U(0,1) ≤ probabilidad E1 ENTONCES
2     Ganador ← E1
3  SINO
4     Ganador ← E2
5  FIN SI
6  RETORNAR Ganador
```

Figura 4-12 Definición del ganador de un encuentro

Este kernel se lanza usando un block con Ecuación (4-6) threads.

$$número\ de\ encuentros = \frac{nT}{2} \quad \textbf{(4-6)}$$

4.5.3.6 Aplicar operadores de imitación y provocación

Después de que los equipos han realizado la jornada, sólo a los equipo ganadores se le aplica los operadores de imitación y provocación descritos en [5]. El operador de imitación se le aplica a los nT jugadores titulares y el de provocación a los nS jugadores sustitutos.

Este kernel se lanza usando Ecuación (4-6) blocks con Ecuación (4-4) threads.

Algoritmo PSLCG(T, nE, nT, nS, HMCR, PAR)

```
1  Inicialización de los jugadores
2  Ordenar jugadores
3  Actualizar el jugador súper estrella
4  PARA t ← 1 HASTA T HACER
5     Fijar los jugadores estrella
6     Calcular la potencia de los equipos
7     PARA ronda ← 1 HASTA 2 HACER
8        PARA jornada ← 1 HASTA nT – 1 HACER
9           Realizar jornada
10          Aplicar operadores de imitación y provocación
11       FIN PARA
12    FIN PARA
13    Ordenar jugadores
14    Actualizar el jugador súper estrella
15 FIN PARA
16 RETORNAR jugador súper estrella
```

Figura 4-13 Algoritmo paralelo PSLCG

4.5.3.7 Actualizar el jugador súper estrella

El jugador súper estrella es aquel con mejor evaluación de la función objetivo entre todos los jugadores de todos los equipos. Este kernel usa un block con un thread y el proceso se muestra en la Figura 4-13 Algoritmo paralelo PSLCG.

4.5.4 Versiones paralelas

Ya que esta fue la última metaheurística paralela que se implementó para este trabajo de grado, se tenía experiencia con los problemas que surgen al implementar un algoritmo usando el modelo de programación de CUDA [41]. En la implementación de SLC paralelo usando la GPU, el principal problema que se identificó para paralelizarlo fue la concurrencia de escritura y lectura (en general todas las metaheurísticas secuenciales tuvieron este problema para realizar la versión paralela, pero en SLC es más evidente); en la versión secuencial el primer equipo se enfrenta con los demás equipos, y luego el segundo con todos hasta terminar los nT equipos, un proceso así no es fácil de paralelizar, por ejemplo para entender el problema imagine que la selección colombiana de fútbol se enfrenta en una jornada simultáneamente con los demás equipos, evidentemente si no se cuenta con el don de la ubicuidad no se podrán realizar los encuentros al mismo tiempo. Entonces se pensó inicialmente que cuando en un thread ganaba Colombia, los demás threads esperaran que se aplicaran los operadores de imitación y provocación y luego se volvía a la ejecución, pero hacer esto no explota el poder de paralelización de la GPU, fue entonces que se decidió implementarlo como se mostró en la Figura 4-11 que es una aproximación natural a una liga de fútbol, y la mejor forma de paralelizarlo omitiendo el problema de concurrencia, existen muchas razones para evitar el problema de concurrencia entre ellas: menor tiempo de ejecución, menor uso de memoria y disminución en el uso de computación híbrida CPU-GPU.

PSCLG es por demás idéntico a su versión secuencial pero con las consideraciones para una versión paralela mencionadas en el párrafo anterior, no se le adicionaron elementos ya que SLC secuencial es paralelizable como SBFPS, a excepción de los caso de MDSFL, SBHS, MopGA donde se tuvieron que realizar cambios para explotar el poder de la GPU, sin embargo para hacer el trabajo de mejora que se realizó en las otras metaheurísticas se probaron diferentes funciones objetivo que fueron las usadas en los artículos [5, 6, 39], de las cuales la mejor fue la del artículo de BFSP [39].

5. EXPERIMENTACIÓN

En este capítulo se muestran los tipos de instancias usadas para el proceso de experimentación. Se presenta el uso de arreglos de cobertura para el afinamiento o ajuste de parámetros, y los resultados y discusión.

5.1 Instancias del problema 0/1 knapsack

Una instancia del problema 0/1 knapsack provee toda la información que se necesita para resolver dicho problema, es decir, la capacidad de la mochila, el número de objetos, y el valor y peso de cada objeto.

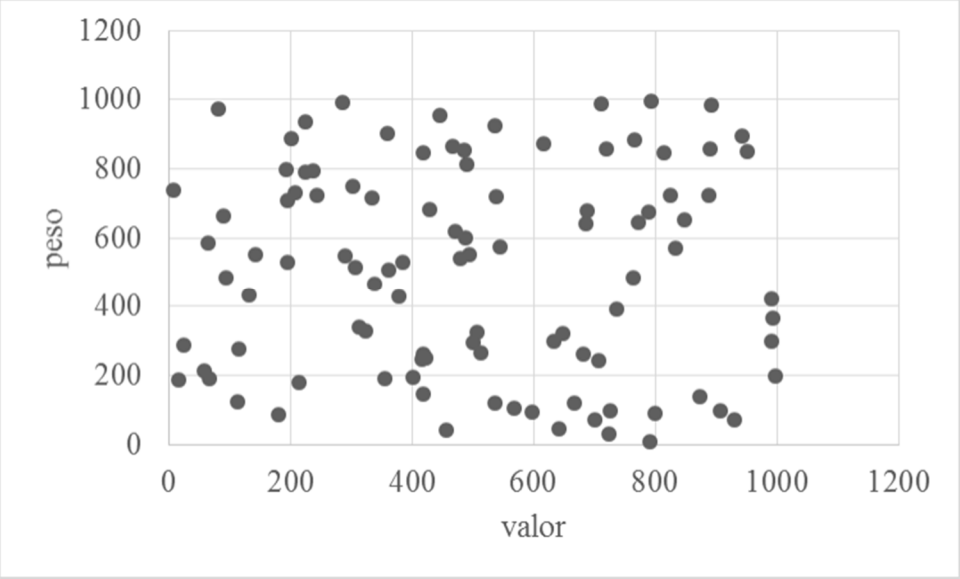

Figura 5-1 Instancias no correlacionadas

5.1.1 Tipos de Instancias del problema 0/1 knapsack

El problema de la mochila tiene diferentes tipos de instancias, David Pisinger [48] considera hasta 14 tipos diferentes, en este trabajo sólo se consideraron 3 que son las más comúnmente usadas en el estado del arte. El tipo de la instancia define la complejidad del problema: instancias no correlacionadas o tipo 1, Figura 5-1; instancias débilmente correlacionadas o tipo 2, Figura 5-2; instancias fuertemente correlacionadas o tipo 3, Figura 5-3.

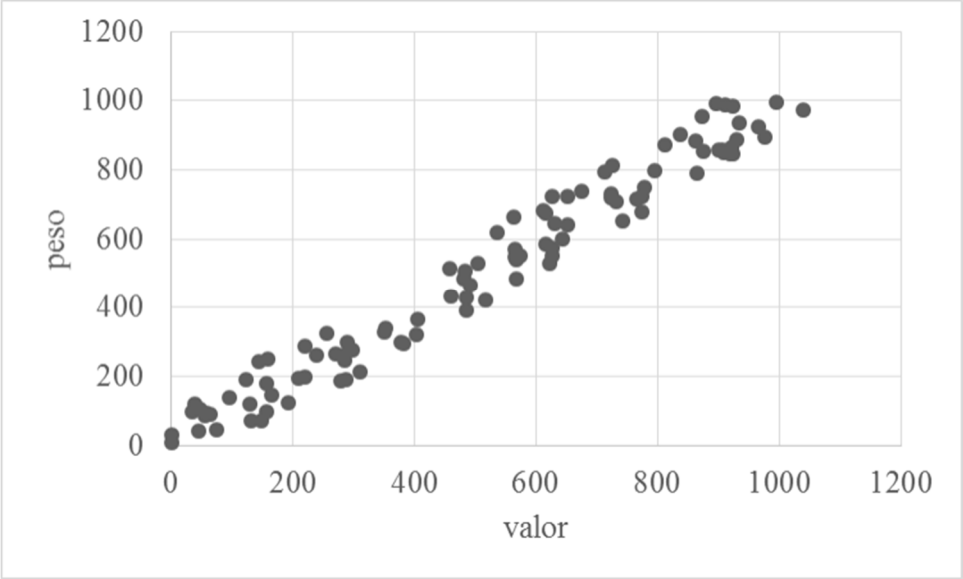

Figura 5-2 Instancias débilmente correlacionadas

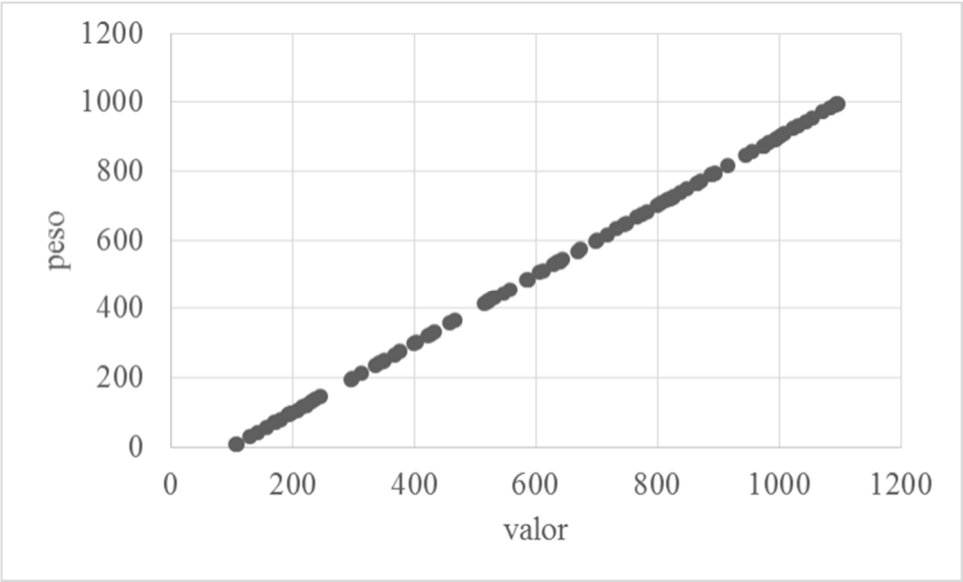

Figura 5-3 Instancias fuertemente correlacionadas

La complejidad de los problemas está dada por su correlación, debido a que, a medida que la correlación va creciendo, es más difícil determinar qué objeto es mejor en la relación a su valor-peso o densidad, ya que esta relación para todos los objetos tiende a un mismo valor, en la medida que la correlación crece.

La densidad [1] como criterio de selección de los mejores objetos para las instancias con más alta correlación se entiende mejor con el siguiente ejemplo: considerando hacer un regalo, se encuentra con dos carros, un carro nuevo y uno usado, ambos del mismo modelo. En este caso la decisión sería sencilla teniendo en cuenta que uno de los carros es inferior al otro, ajustando esta metáfora al concepto de densidad, los carros tienen el mismo volumen, pero el valor del carro usado es inferior por lo tanto el valor de densidad también sería menor. Pero suponiendo que los dos carros son exactamente iguales, en este caso, sus densidades serian iguales y preferir uno sobre el otro, sería cuestión de azar. Así mismo al escoger dos objetos con densidades iguales.

Finalmente, una metaheurística con la capacidad de resolver diferentes tipos de instancias, tiene un mayor rango de acción y por lo tanto una tasa de éxito mayor.

5.2 Afinamiento de parámetros

Cuando se trabaja con metaheurísticas es necesario hacer un ajuste de parámetros, ya que estas son sensibles a los cambios en estos. Del valor de los parámetros depende que la metaheurística mejore o empeore su rendimiento [49].

Para evaluar las metaheurísticas reportadas en el estado del arte se usaron los parámetros recomendados por sus autores como sus parámetros óptimos en la versión secuencial u original de los mismos. En la sección paralela, se realizó un proceso de afinamiento de parámetros para las cinco metaheurísticas debido a que su implementación en paralelo requería de ciertos cambios para aprovechar la arquitectura CUDA.

Debido a las limitaciones de tiempo del proyecto y que los arreglos de cobertura (covering arrays, CAs) permiten el afinamiento de parámetros con amplia cobertura y bajo costos computacional, estos fueron utilizados para soportar este proceso [50].

5.2.1 CAs aplicados al afinamiento de parámetros

Antes de explicar cómo se usaron los CAs para el proceso de afinamiento de parámetros, se presenta una breve descripción de un CA. Un CA es un objeto matemático que se representa como una matriz de la siguiente forma CA (N; t, k, v), donde N es el número de casos de prueba (filas de la matriz), t la fuerza o rango de interacción de las columnas del CA, el cual define la combinación de valores en columnas que deben estar presentes en los casos de prueba del CA, k es el número

de parámetros o columnas, y v es el tamaño del alfabeto o cantidad de valores diferentes en cada columna. N, t, k, y v son números enteros positivos.

5.2.1.1 Ejemplo de afinamiento de parámetros para PSBHSG

Este enfoque de ajuste de parámetros, se usó en todas la metaheurísticas paralelas usando la GPU desarrolladas en este trabajo de grado. El proceso que se describe en esta sección para el ajuste de parámetros fue el mismo para las demás metaheurísticas, los archivos de los CAs usados y el ajuste de parámetros se puede encontrar en el código fuente de este trabajo. A continuación, se describe el proceso.

1) Fijar los parámetros del CA

El parámetro N se fija cuando se ha encontrado el CA. Los valores para los otros parámetros del CA fueron: t igual a 2, k igual a 3 como se mencionó en la sección 4.4.1 Parámetros y corresponde a los parámetros que se van a afinar en el algoritmo, v es igual a 5 (a cada parámetro del algoritmo se le definieron 5 valores). El CA que se busco fue CA (N; 2, 3, 5).

Nota: Para todas las metaheurísticas en este trabajo de grado t y v se mantuvieron con los mismos valores 2 y 5 respectivamente.

2) Búsqueda del CA

Aunque en el Instituto Nacional de Mediciones de Estados Unidos (NIST) se encuentra un sitio web con CAs de libre acceso (http://math.nist.gov/coveringarrays/), ellos no son óptimos y por este motivo se tomaron los CAs construidos en el marco de la tesis de maestría titulada “Algoritmo para la Construcción de Arreglos de Cubrimiento basado en la Mejor Búsqueda Armónica Global y Técnicas de

Optimización Local" que desarrolla la ingeniera Jimena Adriana Timaná bajo la dirección del profesor Carlos Alberto Cobos.

Ya encontrado el CA se puede definir N igual a 25. El CA quedó así CA(25; 2, 3, 5).

3) Definir v valores distintos para cada parámetro

El parámetro v es el tamaño del alfabeto. La forma de definir el alfabeto depende del problema, en este caso se tuvieron en cuenta las consideraciones en el ajuste de parámetros de los autores de las metaheurísticas base para este trabajo de grado [1, 4-6, 39].

El orden de los parámetros en PSBHSG es HMS, NHS y NI; el alfabeto para los parámetros se muestra en la Tabla 5-1

Tabla 5-1 Alfabeto para los parámetros de PSBHSG

HMS =	{5, 10, 20, 30, 40}
NHS =	{10, 15, 20, 25, 30}
NI =	{20, 30, 40, 50, 60}

4) Reemplazar el alfabeto en el CA

Como se muestra en la Tabla 5-2, a partir del CA se generan los casos de afinamiento, en el CA cada columna representa un parámetro y el número dentro de cada celda es el índice del elemento que se toma del alfabeto para fijarlo en los casos del CA, el índice es un valor entre 0 y v – 1. Por ejemplo, columna 1 y fila 1 el valor de la celda es 4, quiere decir que del parámetro HMS se tomara el quinto elemento del alfabeto que es 40 y se fija en la celda de la fila 1 y columna 1 en los casos de afinamiento.

Tabla 5-2 CA y Casos de pruebas para PSBHSG

	CA original [Ing. Jimena]			Casos de Afinamiento		
Fila	HMS	NHS	NI	HMS	NHS	NI
1	4	1	0	40	15	20
2	4	4	1	40	30	30
3	4	2	2	40	20	40
4	4	3	3	40	25	50
5	4	0	4	40	10	60
6	3	4	2	30	30	40
7	3	2	4	30	20	60
8	3	0	3	30	10	50
9	3	1	1	30	15	30
10	3	3	0	30	25	20
11	2	2	3	20	20	50
12	2	0	0	20	10	20
13	2	3	1	20	25	30
14	2	4	4	20	30	60
15	2	1	2	20	15	40
16	1	3	4	10	25	60
17	1	1	3	10	15	50
18	1	4	0	10	30	20
19	1	0	2	10	10	40
20	1	2	1	10	20	30
21	0	0	1	5	10	30
22	0	3	2	5	25	40
23	0	1	4	5	15	60
24	0	2	0	5	20	20
25	0	4	3	5	30	50

5) Análisis de los resultados

Después de ejecutados los casos de pruebas, se siguió con el análisis de los resultados para determinar cuál es el mejor conjunto de parámetros o caso de afinamiento para la metaheurística, esta decisión dependió de los criterios fijados para evaluar la metaheurística.

Para el caso de PSBHSG y las demás metaheurísticas desarrolladas en este proyecto se tienen tres criterios: tasa de éxito, número de evaluaciones de la función objetivo, tiempo de ejecución, y adicionalmente se tomó opcionalmente la memoria usada. Para el primer criterio el mejor caso de prueba es aquel que tenga mayor tasa de éxito; si se presenta el escenario donde los casos de prueba tienen tasas de éxito iguales, se usa el segundo criterio “número de evaluaciones de la función objetivo” y aquel que tenga menor número de evaluaciones promedio será el mejor; si se sigue el empate se tiene el tercer criterio tiempo de ejecución en este escenario el mejor caso de prueba será el de menor tiempo de ejecución promedio, pero si persiste el empate la decisión final se toma con base en el uso del recurso de memoria y aquel que gaste menos recursos será el mejor.

5.3 Resultados y discusión

Esta sección muestra los resultados experimentales y la comparación de las metaheurísticas secuenciales y paralelas (MDSFL [4], MopGA [6], SBFSP, SBHS [1], SLC [5], PBFSPG, PMDSFLG, PMopGAG, PSBHSG y PSLC). Para la ejecución de los experimentos se usó el Framework de pruebas PMG_01KP (parallel metaheuristics using GPU for the 0/1 knapsack problem) desarrollado para este trabajo de grado, el cual se soporta en CUDA 7.5 runtime y fue desarrollado en CUDA C/C++ usando Microsoft Visual Studio Ultimate 2012 en un Intel (R) Core (TM) i5 PC, CPU 650 @ 3.20 GHZ con 8 GB, NVIDIA GeForce GTX 560, y Windows 8 64-bit.

5.3.1 Criterios de evaluación de las metaheurísticas

Se usaron tres criterios de evaluación valor óptimo, tiempo de ejecución y número de evaluaciones de la función objetivo. En este documento solo se muestra los resultados de los valores óptimos, los resultados de los demás criterios los pueden encontrar en el Anexo B.

5.3.2 Instancias de prueba

Se seleccionaron 21 instancias disponibles en [33] y sus archivos fueron adaptados al formato usado en este trabajo. De las 21 instancias se tienen siete (7) diferentes dimensiones y tres (3) tipos de instancias (no correlacionadas, débilmente correlacionadas y fuertemente correlacionadas), las dimensiones fueron medianas (100 <= n < 1000) 100, 200 y 500 y grandes [51] (n >= 1000) 1000, 2000, 5000 y 10000.

5.3.3 Resultados en el afinamiento de parámetros

La Tabla 5-3 muestra los parámetros de las metaheurísticas secuenciales tal y como fueron propuestos por los autores; para propósitos de esta investigación, se unificaron los criterios de parada a un tiempo máximo de ejecución. La Tabla 5-4 muestran el resultado final del afinamiento de parámetros usando el enfoque de CA para las metaheurísticas paralelas propuestas, los resultados de los experimentos del afinamiento de parámetros los puede encontrar en el Anexo B.

Tabla 5-3 Parámetros metaheurísticas secuenciales

Metaheurística	Parámetros
MDSFL [4]	P = 200; m = 10; it = 10; iMáx = n/a; alpha = 0.4; pm = 0.06
MopGA [6]	Pm = 0.15; Pi = 0.2; N = 50; G = 1500
SBFSP	T = n/a; N = 60; M = 50; p_m = 0.30; p_b = 0.50; iMáx=10
SBHS [1]	HMS = 5; HMSCR = 0.95; NI = n/a
SLC [5]	S=n/a, nE=10; nT=10; nS=10; HMC =0.1; PAR=0.05; λ = n/a

Tabla 5-4 Parámetros metaheurísticas paralelas

Metaheurística	Parámetros
PBFSPG	T = n/a; N = 60; M = 50; p_m = 0.30; p_b = 0.50; iMáx=10
PMDSFLG	P = 800; m = 10; it = 25; iMax= n/a; α = 0.4; pm = 0.03; N = 45
PMopGAG	p_m = 0.25; p_i = 0.25; N = 120; g_p = 100; M = 70
PSBHSG	HMS = 20; T = n/a; NHS = 10; NI = 20
PSLCG	S=n/a; nE=32; nT=10; nS=18; HMCR=0.15; PAR=0.125; λ = n/a

5.3.4 Resultados de las metaheurísticas secuenciales

La evaluación de los casos secuenciales se llevó a cabo de la siguiente manera: se corrieron cada uno de los problemas 30 veces y se les dio un tiempo máximo de ejecución de 30 minutos a cada algoritmo por cada ejecución. Por cuestiones de tiempo y observando que los problemas más grandes requieren de más tiempo, se les dio un límite de error para encontrar la solución del 10% como mínimo (3 de las 30 corridas). En caso de no alcanzar este límite, no se les dio la oportunidad de correr problemas de mayor dimensión.

La Tabla 5-5, Tabla 5-6 y Tabla 5-7 muestran para el criterio valor optimo la tasa de éxito (TE), el mejor valor, el peor valor, la mediana (Me), la media y la desviación estándar poblacional (DesvEst.P), como se puede observar SBHS y MopGA no fueron capaces de resolver los problemas de 1000 objetos en adelante. Así mismo se puede observar que SLC y MDSFL no resolvieron problemas de 2000 objetos en adelante. Por otra parte, SBFSP fue capaz de encontrar respuestas con tasas de éxito del 100% para todos los problemas fuertemente correlacionados. Para los tipos no correlacionados y débilmente correlacionados encuentra el óptimo para instancias de hasta 5000 objetos. Cabe resaltar la fuerte competencia de MDSFL frente a SBFSP, el cual fue capaz de superarlo en tiempo, en algunas instancias donde ambos tienen una tasa de éxito del 100%, además de superar la tasa de éxito de SBFSP en el problema de 100 objetos débilmente correlacionados que se observa en la Tabla 5-6.

Tabla 5-5 Resultados secuenciales para el criterio valor óptimo para instancias no correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
100	SBFSP	1	9147	9147	9147	9147	0
	MDSFL	1	9147	9147	9147	9147	0
	SBHS	0,9333	9147	8929	9147	9132,4667	54,3788
	SLC	1	9147	9147	9147	9147	0
	MopGA	0,8333	9147	8929	9147	9110,6667	81,2438
200	SBFSP	1	11238	11238	11238	11238	0
	MDSFL	1	11238	11238	11238	11238	0
	SBHS	0,6667	11238	10900	11238	11195,4333	88,9609
	SLC	1	11238	11238	11238	11238	0
	MopGA	0,3333	11238	10722	11223	11131,2667	151,7447
500	**SBFSP**	**1**	**28857**	**28857**	**28857**	**28857**	**0**
	MDSFL	0,1	28857	28589	28834	28774,6667	81,0614
	SBHS	0	28321	26493	27592,5	27591,2333	526,218
	SLC	0,9667	28857	28834	28857	28856,2333	4,1286
	MopGA	0	28834	27699	28577,5	28524,2667	229,1013
1000	**SBFSP**	**1**	**54503**	**54503**	**54503**	**54503**	**0**
	MDSFL	0	54354	52730	53995	53871	362,8496
	SBHS	-	-	-	-	-	-
	SLC	0,1333	54503	54440	54482	54474,4	17,7042
	MopGA	-	-	-	-	-	-
2000	**SBFSP**	**0,8667**	**110625**	**110597**	**110625**	**110622,733**	**7,0377**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
5000	**SBFSP**	**0,0333**	**276457**	**276456**	**276456**	**276456,033**	**0,1795**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
10000	**SBFSP**	**0**	**563643**	**563588**	**563631**	**563628,433**	**10,2914**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-

Tabla 5-6 Resultados secuenciales para el criterio valor óptimo para instancias débilmente correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
	SBFSP	0,9	1514	1513	1514	1513,9	0,3
	MDSFL	**1**	**1514**	**1514**	**1514**	**1514**	**0**
100	SBHS	0,1	1514	1481	1513	1505,7333	10,3921
	SLC	0,9667	1514	1502	1514	1513,6	2,1541
	MopGA	0,5333	1514	1452	1514	1501,0667	17,3262
	SBFSP	1	1634	1634	1634	1634	0
	MDSFL	1	1634	1634	1634	1634	0
200	SBHS	0,7667	1634	1604	1634	1630,6333	7,2777
	SLC	0,3667	1634	1615	1622,5	1625,2667	7,2015
	MopGA	0	1620	1313	1579	1551,4667	71,0079
	SBFSP	**1**	**4566**	**4566**	**4566**	**4566**	**0**
	MDSFL	0,3333	4566	4542	4555	4556,4333	7,9987
500	SBHS	0	4551	4385	4500,5	4497,9333	35,2977
	SLC	0,2	4566	4523	4559	4554,4	11,2268
	MopGA	0	4506	3930	4238,5	4232,2667	119,5028
	SBFSP	**1**	**9052**	**9052**	**9052**	**9052**	**0**
	MDSFL	0	8748	8589	8652	8653,4333	40,1395
1000	SBHS	-	-	-	-	-	-
	SLC	0,0333	9052	9010	9029,5	9029,6333	10,5182
	MopGA	-	-	-	-	-	-
	SBFSP	**0,2333**	**18051**	**18048**	**18049**	**18049,2333**	**1,0546**
	MDSFL	-	-	-	-	-	-
2000	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
	SBFSP	**0,2333**	**44356**	**44352**	**44354**	**44353,9667**	**1,2776**
	MDSFL	-	-	-	-	-	-
5000	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
	SBFSP	**0**	**90203**	**90199**	**90202**	**90201,7**	**0,9713**
	MDSFL	-	-	-	-	-	-
10000	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-

Tabla 5-7 Resultados secuenciales para el criterio valor óptimo para instancias fuertemente correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
100	SBFSP	1	2397	2397	2397	2397	0
	MDSFL	**1**	**2397**	**2397**	**2397**	**2397**	**0**
	SBHS	0,9667	2397	2390	2397	2396,7667	1,2565
	SLC	0,4	2397	2296	2396	2353,4667	49,4474
	MopGA	0,2333	2397	2292	2297	2336,4333	49,1221
200	SBFSP	1	2697	2697	2697	2697	0
	MDSFL	**1**	**2697**	**2697**	**2697**	**2697**	**0**
	SBHS	1	2697	2697	2697	2697	0
	SLC	0,8667	2697	2696	2697	2696,8667	0,3399
	MopGA	0,1333	2697	2197	2597	2573,3667	102,1536
500	**SBFSP**	**1**	**7117**	**7117**	**7117**	**7117**	**0**
	MDSFL	0,4	7117	7017	7017	7060,3	49,5158
	SBHS	0,0333	7117	6817	7017	6993,2	66,5429
	SLC	0,1	7117	7017	7017	7027	30
	MopGA	0	6517	5417	5817	5923,6667	336,5841
1000	**SBFSP**	**1**	**14390**	**14390**	**14390**	**14390**	**0**
	MDSFL	0,0333	14390	14190	14290	14280	39,5811
	SBHS	-	-	-	-	-	-
	SLC	0	14290	14190	14190	14230	48,9898
	MopGA	-	-	-	-	-	-
2000	**SBFSP**	**1**	**28919**	**28919**	**28919**	**28919**	**0**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
5000	**SBFSP**	**1**	**72505**	**72505**	**72505**	**72505**	**0**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-
10000	**SBFSP**	**1**	**146919**	**146919**	**146919**	**146919**	**0**
	MDSFL	-	-	-	-	-	-
	SBHS	-	-	-	-	-	-
	SLC	-	-	-	-	-	-
	MopGA	-	-	-	-	-	-

5.3.5 Resultados de las metaheurísticas paralelas usando la GPU

La evaluación en los casos paralelos se llevó de forma similar a la evaluación secuencial. En este caso se corrieron 30 veces con un tiempo de 40 segundos por cada corrida, el objetivo de este tiempo era demostrar la capacidad de la arquitectura CUDA para acortar los tiempos de ejecución de los algoritmos.

En la Tabla 5-9, Tabla 5-10 y Tabla 5-11 , se puede observar que PBFSPG fue capaz de resolver todos los problemas con tasa de éxito 100% exceptuando 4 instancias (sin correlación de 10000 y débilmente correlacionadas de 2000, 5000 y 10000). A pesar de la ayuda de la arquitectura CUDA, PMopGAG y PSLCG tuvieron un bajo desempeño, con una tasa promedio de éxito del 38% y del 30% respectivamente. Por otro lado, PMDSFLG, PBFSPG y PSBHSG tuvieron tasa promedio de éxito del 93.4%, del 93% y del 81% respectivamente.

Es preciso observar que los resultados evidencian una clara competencia entre PBFSPG y PMDSFLG. Los dos algoritmos tienen una tasa de éxito similar, aunque en la mayoría de los casos, PMDSFLG supera a PBFSPG con tiempos de ejecuciones menores en algunas instancias.

Tabla 5-8 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias no correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
100	PBFSPG	1	9147	9147	9147	9147	0
	PMDSFLG	1	9147	9147	9147	9147	0
	PSBHSG	**1**	**9147**	**9147**	**9147**	**9147**	**0**
	PSLCG	1	9147	9147	9147	9147	0
	PMopGAG	1	9147	9147	9147	9147	0
200	PBFSPG	1	11238	11238	11238	11238	0
	PMDSFLG	**1**	**11238**	**11238**	**11238**	**11238**	**0**
	PSBHSG	1	11238	11238	11238	11238	0
	PSLCG	0,9667	11238	11223	11238	11237,5	2,6926
	PMopGAG	1	11238	11238	11238	11238	0
500	PBFSPG	1	28857	28857	28857	28857	0
	PMDSFLG	**1**	**28857**	**28857**	**28857**	**28857**	**0**
	PSBHSG	0,9333	28857	28834	28857	28855,4667	5,7372
	PSLCG	0,5667	28857	28746	28857	28841,7667	23,7889
	PMopGAG	0,6667	28857	28794	28857	28846	17,088
1000	PBFSPG	1	54503	54503	54503	54503	0
	PMDSFLG	**1**	**54503**	**54503**	**54503**	**54503**	**0**
	PSBHSG	1	54503	54503	54503	54503	0
	PSLCG	0	42304	29198	37249	37142,3333	3130,5178
	PMopGAG	0	54377	53398	54068	54023,2333	229,649
2000	**PBFSPG**	**1**	**110625**	**110625**	**110625**	**110625**	**0**
	PMDSFLG	0,8667	110625	110619	110625	110624,2	2,0396
	PSBHSG	0,9333	110625	110619	110625	110624,6	1,4967
	PSLCG	0	-11526319	-13685654	-12316937,5	-12369473,7	535803,984
	PMopGAG	0	-6,247E+24	-8,1476E+24	-7,7561E+24	-7,6913E+24	3,4705E+23
5000	PBFSPG	1	276457	276457	276457	276457	0
	PMDSFLG	**1**	**276457**	**276457**	**276457**	**276457**	**0**
	PSBHSG	1	276457	276457	276457	276457	0
	PSLCG	0	-82242992	-84520776	-83576868	-83499777,1	543865,843
	PMopGAG	0	-5,906E+25	-8,9264E+25	-8,8498E+25	-8,7467E+25	5,2933E+24
10000	PBFSPG	0,9667	563647	563636	563647	563646,633	1,9746
	PMDSFLG	**1**	**563647**	**563647**	**563647**	**563647**	**0**
	PSBHSG	0	547258	541309	544193,5	544339,1	1323,0283
	PSLCG	0	-2,07209E8	-210423856	-209389936	-209164758	876227,75
	PMopGAG	0	-1,176E+26	-2,1972E+26	-2,1862E+26	-2,1522E+26	1,8144E+25

Tabla 5-9 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias débilmente correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
100	PBFSPG	1	1514	1514	1514	1514	0
	PMDSFLG	1	1514	1514	1514	1514	0
	PSBHSG	**1**	**1514**	**1514**	**1514**	**1514**	**0**
	PSLCG	1	1514	1514	1514	1514	0
	PMopGAG	1	1514	1514	1514	1514	0
200	PBFSPG	1	1634	1634	1634	1634	0
	PMDSFLG	**1**	**1634**	**1634**	**1634**	**1634**	**0**
	PSBHSG	1	1634	1634	1634	1634	0
	PSLCG	0,8333	1634	1620	1634	1631,9667	4,6796
	PMopGAG	0,8333	1634	1624	1634	1633,2667	2,1281
500	PBFSPG	1	4566	4566	4566	4566	0
	PMDSFLG	**1**	**4566**	**4566**	**4566**	**4566**	**0**
	PSBHSG	1	4566	4566	4566	4566	0
	PSLCG	0,0333	4566	4507	4543	4541,3	15,1595
	PMopGAG	0,5667	4566	4516	4566	4560,2333	9,9488
1000	PBFSPG	1	9052	9052	9052	9052	0
	PMDSFLG	**1**	**9052**	**9052**	**9052**	**9052**	**0**
	PSBHSG	1	9052	9052	9052	9052	0
	PSLCG	0	7416	5514	6707	6606,8	388,0111
	PMopGAG	0	9015	8726	8935,5	8925	67,9304
2000	PBFSPG	0,8333	18051	18050	18051	18050,8333	0,3727
	PMDSFLG	**1**	**18051**	**18051**	**18051**	**18051**	**0**
	PSBHSG	1	18051	18051	18051	18051	0
	PSLCG	0	-11839400	-14986971	-13072547	-13114351	678983,093
	PMopGAG	0	-6,247E+24	-8,1037E+24	-7,7604E+24	-7,6986E+24	3,2319E+23
5000	PBFSPG	0,5333	44356	44355	44356	44355,5333	0,4989
	PMDSFLG	**0,7667**	**44356**	**44355**	**44356**	**44355,7667**	**0,423**
	PSBHSG	0,3333	44356	44355	44355	44355,3333	0,4714
	PSLCG	0	-83960904	-87043104	-85615252	-85564600	650774,204
	PMopGAG	0	-5,9062E+25	-8,9156E+25	-8,8403E+25	-8,7341E+25	5,2713E+24
10000	**PBFSPG**	**0,2**	**90204**	**90203**	**90203**	**90203,2**	**0,4**
	PMDSFLG	0	90203	90203	90203	90203	0
	PSBHSG	0	89738	89405	89531	89541,3	86,2265
	PSLCG	0	-210390240	-214475376	-212229248	-212402919	960698,818
	PMopGAG	0	-1,1762E+26	-2,1978E+26	-2,1883E+26	-2,1531E+26	1,8157E+25

Tabla 5-10 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias fuertemente correlacionadas

n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
100	PBFSPG	1	2397	2397	2397	2397	0
	PMDSFLG	**1**	**2397**	**2397**	**2397**	**2397**	**0**
	PSBHSG	1	2397	2397	2397	2397	0
	PSLCG	0,9333	2397	2297	2397	2390,3333	24,9444
	PMopGAG	1	2397	2397	2397	2397	0
200	PBFSPG	1	2697	2697	2697	2697	0
	PMDSFLG	**1**	**2697**	**2697**	**2697**	**2697**	**0**
	PSBHSG	1	2697	2697	2697	2697	0
	PSLCG	1	2697	2697	2697	2697	0
	PMopGAG	1	2697	2697	2697	2697	0
500	PBFSPG	1	7117	7117	7117	7117	0
	PMDSFLG	**1**	**7117**	**7117**	**7117**	**7117**	**0**
	PSBHSG	1	7117	7117	7117	7117	0
	PSLCG	0	7017	6917	7017	7003,6667	33,9935
	PMopGAG	0,9	7117	7017	7117	7107	30
1000	PBFSPG	1	14390	14390	14390	14390	0
	PMDSFLG	**1**	**14390**	**14390**	**14390**	**14390**	**0**
	PSBHSG	1	14390	14390	14390	14390	0
	PSLCG	0	11985	10562	11528	11447,7	377,5325
	PMopGAG	0	14290	13990	14190	14182	85,0733
2000	PBFSPG	1	28919	28919	28919	28919	0
	PMDSFLG	**1**	**28919**	**28919**	**28919**	**28919**	**0**
	PSBHSG	1	28919	28919	28919	28919	0
	PSLCG	0	-14072133	-17199444	-14950091	-15095529,6	640016,97
	PMopGAG	0	-6,042E+24	-7,8576E+24	-7,5726E+24	-7,4936E+24	3,2697E+23
5000	PBFSPG	1	72505	72505	72505	72505	0
	PMDSFLG	**1**	**72505**	**72505**	**72505**	**72505**	**0**
	PSBHSG	0,8333	72505	72504	72505	72504,8333	0,3727
	PSLCG	0	-99519616	-103065472	-101442648	-101372182	817264,509
	PMopGAG	0	-5,8929E+25	-8,8461E+25	-8,7757E+25	-8,6765E+25	5,1882E+24
10000	PBFSPG	1	146919	146919	146919	146919	0
	PMDSFLG	**1**	**146919**	**146919**	**146919**	**146919**	**0**
	PSBHSG	0	143289	141700	142180,5	142188,667	302,3879
	PSLCG	0	-250693584	-253703888	-252481152	-252359013	775466,775
	PMopGAG	0	-1,1496E+26	-2,1845E+26	-2,1705E+26	-2,1369E+26	1,8351E+25

En la Tabla 5-11, se muestran los resultados de las instancias más difíciles (las cuales PBFSPG no pudo resolver en 40 segundos). Para esta evaluación se tomó como parámetro el tiempo límite que le tomaba a PBFSPG para resolverlas y a partir de este tiempo evaluar el resto de algoritmos. Como se puede observar los algoritmos que tuvieron bajo rendimiento, no tuvieron ningún cambio al respecto, pero PBFSPG fue capaz de aumentar su tasa de éxito al 100% en todas las instancias.

Tabla 5-11 Resultados paralelos usando la GPU para el criterio valor óptimo para instancias difíciles

Tipo	T. Máx. (min)	n	Metaheurística	TE	Mejor	Peor	Me	Media	DesvEst.P
1	2	10000	PBFSPG	1	563647	563647	563647	563647	0
			PMDSFLG	1	563647	563647	563647	563647	0
			PSBHSG	0,8333	563647	563643	563647	563646,467	1,2311
			PSLCG	0	-173810304	-179406448	-176343408	-176336008	1364317
			PMopGAG	0	-1,1577E+26	-1,7609E+26	-1,7486E+26	-1,7287E+26	1,0622E+25
2	3	2000	PBFSPG	1	18051	18051	18051	18051	0
			PMDSFLG	1	18051	18051	18051	18051	0
			PSBHSG	1	18051	18051	18051	18051	0
			PSLCG	0	15270	-1130446,25	12585	-129333,998	362985,834
			PMopGAG	0	18022	17501	17917,5	17910,1	98,0083
	9	5000	**PBFSPG**	**1**	**44356**	**44356**	**44356**	**44356**	**0**
			PMDSFLG	0,7667	44356	44355	44356	44355,7667	0,423
			PSBHSG	0,3333	44356	44355	44355	44355,3333	0,4714
			PSLCG	0	-7149103,5	-10505060	-8627772	-8885859,17	774371,608
			PMopGAG	0	41553	30820	40584	40278,2667	1825,1745
	20	10000	**PBFSPG**	**1**	**90204**	**90204**	**90204**	**90204**	**0**
			PMDSFLG	0	90203	90203	90203	90203	0
			PSBHSG	0,1667	90204	90203	90203	90203,1667	0,3727
			PSLCG	0	-36130352	-45105232	-39820712	-39778069,9	2060595,69
			PMopGAG	0	59727	-2,2128E+25	55455	-7,4384E+23	3,9711E+24

Un comportamiento interesante se puede observar en la instancia débilmente correlacionada de 10000 objetos, donde BFSP fue capaz de resolverla con una tasa de éxito del 100% mientras MDSFL, tuvo una tasa de éxito de 0% con el mismo tiempo, lo cual indica que MDSFL se queda atrapado en ciertos óptimos locales.

5.3.6 Análisis de la aceleración de PBFSPG

Entre las Tabla 5-5 a Tabla 5-11 se observa que PBFSPG obtuvo una mejora en la calidad de los resultados respecto a su versión secuencial SBFSP. En esta sección se muestra que PBFSPG también mejoró en el tiempo de ejecución.

Tabla 5-12 Speed-up de SBFSP a PBFSPG

Tipo	n	Paralelo	Secuencial	Speed-up	%Reducción
	100	0,0073	0,0359	4,9178	79,6657
	200	0,0203	0,0729	3,5911	72,1536
	500	4,7122	71,408	15,1539	93,4010
1	1000	0,1229	6,8545	55,773	98,2070
	2000	-	-	-	-
	5000	-	-	-	-
	10000	-	-	-	-
	100	-	-	-	-
	200	0,0193	0,0792	4,1036	75,6313
	500	0,2896	3,1241	10,7876	90,7301
2	1000	0,2401	54,9797	228,9867	99,5633
	2000	-	-	-	-
	5000	-	-	-	-
	10000	-	-	-	-
	100	0,0089	0,0542	6,0899	83,5793
	200	0,0161	0,0526	3,2671	69,3916
	500	0,0438	0,3761	8,5868	88,3542
3	1000	0,0938	1,8334	19,5458	94,8838
	2000	0,2636	8,0587	30,5717	96,7290
	5000	1,0433	62,0411	59,4662	98,3184
	10000	2,6256	333,3385	126,9571	99,2123
					%Reducción Promedio = 88,56

En la Tabla 5-12 se muestra el tiempo de ejecución promedio en segundos de las instancias que tuvieron tasa de éxito del 100% con el tiempo máximo que se asignó en la experimentación para la versión paralela (PBFSPG) y la versión secuencial (SBFSP), la columna Speed-up o aceleración se calcula usando la Ecuación (5-1) y

la columna porcentaje de reducción (%Reducción) se calcula usando la Ecuación (5-2).

$$Speed\ up = \frac{tiempo\ secuencial}{tiempo\ paralelo} \quad \textbf{(5-1)}$$

$$\%Reduccion = \frac{tiempo\ secuencial \quad tiempo\ paralelo}{tiempo\ secuencial} \times 100 \quad \textbf{(5-2)}$$

La Figura 5-4 muestra la aceleración de PBFSPG para las instancias no correlacionadas desde n = 100 hasta n = 1000, se aprecia que a medida que la dimensión crece el uso de una implementación paralela usando la GPU se ejecuta mucho más rápidamente.

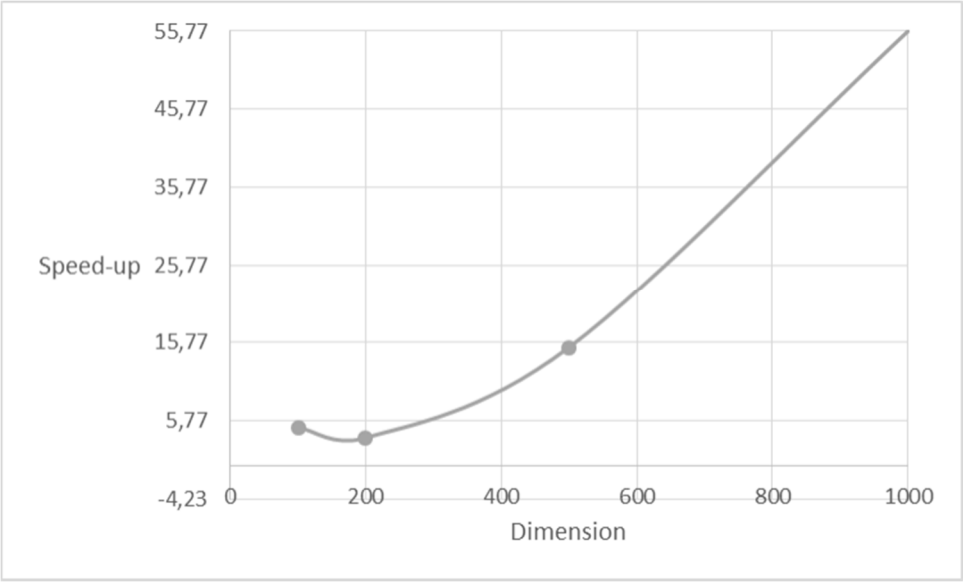

Figura 5-4 Speed-up PBFSPG instancias no correlacionadas

La Figura 5-5 muestra la aceleración de PBFSPG para las instancias débilmente correlacionadas para n = 200, 500 y 1000. Igual que para las instancias no correlacionadas a medida que la dimensión del problema crece el uso PBFSPG muestra mayor rapidez, inclusive en estas instancias se logró una mejora notoria

sobre las instancias anteriores, un speed up máximo de 228.98 mientras que en las anteriores fue de 55.77. En este tipo de instancias es donde se encuentra una mayor mejora en la velocidad.

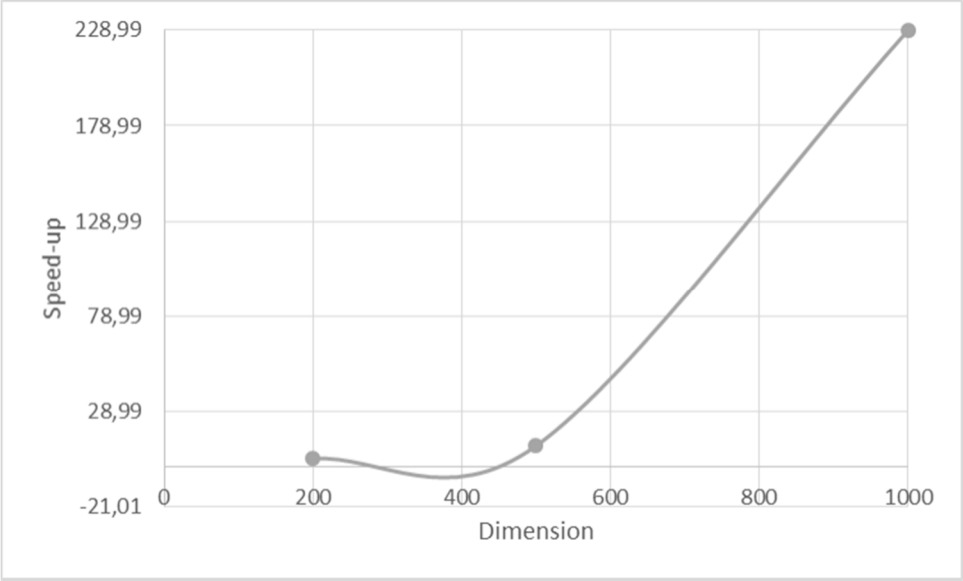

Figura 5-5 Speed-up PBFPSG instancias débilmente correlacionadas

La Figura 5-6 muestra el speed-up de instancias fuertemente correlacionadas para n de 100 a 10000. Los resultados muestran un comportamiento similar al de los otros tipos de instancias, logrando ser dos veces más rápidos en estas instancias que en las primeras (no correlacionadas).

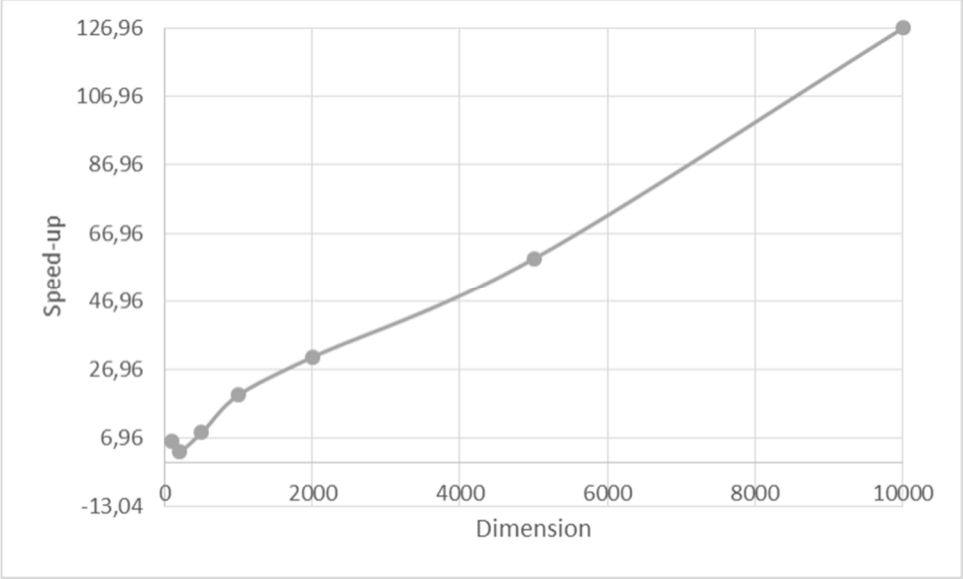

Figura 5-6 Speed-up PBFSPG instancias fuertemente correlacionadas

Un análisis similar para las otras metaheurísticas no se pudo llevar a cabo ya que la tasa de éxito para las metaheurísticas secuenciales no fue de 100% en la mayoría de los casos, por esta razón no existía una muestra confiable, sobre la cual realizar dicho análisis.

Haciendo la comparación entre PBFSPG y SBFSP. PBFSPG reduce el tiempo de ejecución de SBFSP en promedio un 88.56%, Tabla 5-12, quedando demostrado que se hizo una adecuada paralelización de la metaheurística y muestra la superioridad de usar la arquitectura CUDA para este problema. En general para las implementaciones paralelas usando la GPU la calidad de las repuestas mejora y el tiempo de ejecución se reduce, ver Anexo B

6. CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO

6.1 Conclusiones

En este trabajo se implementaron cinco (5) metaheurísticas paralelas que se ejecutan en un entorno híbrido CPU-GPU para resolver el problema binario de la mochila (problema 0/1 Knapsack) usando dos aproximaciones, la primera enfocada a la penalización de las soluciones no factibles (PMopGA y PSLC) y la segunda orientada a la reparación de dichas soluciones no factibles (PBFSPG, PMDSFLG y PSBHS).

Las implementaciones paralelas superaron en rendimiento a sus contrapartes secuenciales. La metaheurística paralela que reportó mejores resultados respecto al tiempo de ejecución fue PMDSFLG, y la metaheurística paralela que reportó mejores resultados respecto a la tasa de éxito fue PBFSPG, presupuesto en este trabajo de grado. Adicionalmente, usando como base PBFSPG se implementó una nueva metaheurística secuencial SBFSP la cual obtuvo un buen rendimiento para las instancias de grandes dimensiones (n >= 1000), superando a las otras metaheurísticas del estado del arte con las que se comparó.

Aunque todas las implementaciones paralelas superaron a sus bases secuenciales, no es correcto afirmar que todo algoritmo secuencial debe tender a la paralelización; ya que un rendimiento mayor o menor usado CUDA está dado por el nivel de paralelización que tenga el algoritmo y los recursos que este use, si un algoritmo tiene un bajo nivel de paralelización y un alto uso de recursos (como la memoria) la paralelización podría obtener un rendimiento menor que su versión secuencial.

Para este trabajo de grado se desarrollaron implementaciones secuenciales y paralelas, para realizar la comparación. Sin embargo, si se ha decidido usar CUDA

para realizar una solución paralela, la implementación debe ser diseñada desde las más tempranas etapas para adaptarse al modelo de programación de CUDA.

Las metaheurísticas que siguieron el enfoque de penalización obtuvieron el rendimiento más bajo, mientras que la que usaron el enfoque de reparación obtuvieron los mejores resultados.

La ventaja del enfoque de reparación sobre el de penalización radica en la reducción del espacio de búsqueda que realizan las primeras al buscar sólo en el espacio de soluciones factibles. A medida que los problemas se hacen más grandes su espacio de búsqueda crece exponencialmente, por ejemplo, para un problema de 10000 objetos se tiene un espacio de búsqueda de 2^{10000} combinaciones posibles, pero si se toman sólo las combinaciones factibles ese espacio se reduce en función del peso de los objetos y la capacidad de la mochila a una fórmula igual a $\sum_{i=1}^{m} \binom{n}{i}$ que es mucho menor, ya que en muchos casos m es mucho menor que n (total de objetos).

Para el ajuste de parámetros se usó el enfoque de arreglos de cobertura, el cual es una aproximación real para el ajuste de parámetros; y no una simple exploración de prueba y error para definir cuál es el mejor conjunto de valores para los parámetros, como generalmente se realiza en este tipo de investigaciones. Este enfoque permite una amplia cobertura de los valores de los parámetros y un bajo coste computacional. Para proyectos con limitaciones de tiempo como este trabajo de grado, fue la mejor opción ya que en el caso de usar CAs óptimos se tienen el mínimo número de casos de prueba reduciendo así el tiempo de experimentación, el cual podría ser restrictivo para el proyecto si se usaban otras técnicas como redes neuronales.

Para la experimentación y comparación de los diferentes algoritmos, se usaron 21 problemas binarios de la mochila con diferente dificultad (no correlacionados,

débilmente correlacionados y fuertemente correlacionados) suministrados por un tercero. Al usar estos problemas propuestos por terceros se evitó sesgar los resultados de la investigación.

Una de las grandes diferencias de PBFSPG y de SBFSP respecto a las otras metaheurísticas, es el uso de tres procedimientos de reparación (simple, basado en densidad y basado en valor), los cuales le dan la capacidad de resolver diferentes tipos de instancias con diferentes dimensiones pequeñas, medianas y grandes.

La implementación de una metaheurística paralela para CPU-GPU es un trabajo más complejo y requiere más tiempo que su desarrollo en forma secuencial o paralelo sólo para CPU. Se debe identificar los elementos a paralelizar para el diseño e implementación de la metaheurística paralela para CPU-GPU. Además, en el proceso de diseño se deben adaptar adecuadamente aspectos de la arquitectura CUDA, puesto que, según la naturaleza de muchas metaheurísticas, es posible que la implementación directa no este aprovechando completamente la capacidad de las GPUs.

El Framework desarrollado (PMG_01KP) para la implementación de las metaheurísticas secuenciales y paralelas para CPU-GPU se entrega como una ayuda para los investigadores que deseen estudiar el problema binario de la mochila. La principal característica de este Framework es que el investigador se podrá concentrar exclusivamente en la implementación de la metaheurística, y olvidarse de la lectura y escritura de los archivos, configuración de los experimentos, y en un pequeño grado de la asignación y liberación de memoria que suele ser uno de los principales inconvenientes para los desarrolladores de CUDA en C/C++. El código fuente y la documentación relacionados con este trabajo se encuentran disponibles en https://bitbucket.org/jaoruiz/01kp_solution.

6.2 Trabajo futuro

En este trabajo de grado sólo se usaron 3 tipos de problemas binarios de la mochila (no correlacionados, débilmente correlacionados y fuertemente correlacionados), de los 14 documentados en [48]. Para evaluar la capacidad de PBFSPG y de los otros algoritmos del estado del arte presentados en este documento se debe ampliar el tipo de problemas y experimentar con ellos para comprobar si resuelven esos problemas, sino realizar modificaciones en el algoritmo propuesto (PBFSPG) para la creación de nudos y probar diferentes configuraciones para la generación de vecindario.

Basado en No-Free Lunch Theorem es sabido que algunas metaheurísticas tienen ventaja frente a otras en resolver ciertos tipos de problemas. Es necesario investigar el comportamiento de estas instancias y buscar un algoritmo o vecindario que se adapte mejor a cada tipo de instancia, ya sea haciendo que el algoritmo identifique el tipo de instancia que se está enfrentando o su comportamiento sea oscilatorio, es decir que vaya adaptando sus métodos a medida que identifica cuales son mejores para cada instancia. Para lo anterior, se hace promisorio el uso de un enfoque basado en Multiple Offspring Sampling (MOS).

El problema de estudio para este trabajo de grado fue el problema 0/1 Knapsack, y para esto se diseñaron e implementaron diferentes metaheurísticas paralelas para entornos paralelos CPU-GPU (PBFSPG, PMDSFLG, PMopGAG, PSBHSG, PSLCG). Dado que todas estas metaheurísticas han sido diseñados para resolver problemas genéricos [52], se pueden usar para resolver otros problemas binarios, pero también continuos y discretos. Se espera que el GTI use los diseños paralelos para CPU-GPU en otros problemas como la generación automática de resúmenes de texto, Job Schedule, Timetabling, Facility Location, Graph Coloring, TSP y SAT.

Actualmente existen implementaciones de alto rendimiento de metaheurísticas paralelas usando la GPU pero sólo para problemas continuos [42, 51, 53], si bien la implementación de PBFSPG reduce el tiempo de ejecución, aún es posible realizar un diseño e implementación de alto rendimiento de esta metaheurística, y de esta forma resolver problemas de grandes dimensiones en menor tiempo.

6.3 Recomendaciones

El número de aplicaciones que usan implementaciones paralelas híbridas CPU-GPU está creciendo. En [54] hacen un listado de las áreas de aplicación, que incluye aplicaciones en Finanzas, Análisis de datos, Defensa e inteligencia, Deep Learning y Machine Learning, entre otros. Esta tecnología cada día se usa más por su bajo coste, alto rendimiento y facilidad de desarrollo. Para brindar la oportunidad de realizar nuevas investigaciones en esta área, el Departamento de Sistemas debe considerar adquirir equipos con GPUs, o realizar alianzas con empresas como Amazon o Microsoft por mencionar algunas, para usar los servicios de virtualización de instancias con GPUs en la nube como AWS (Amazon Web Service) o Windows Azure, ya que actualmente no se dispone de este tipo de equipos y servicios en el Departamento.

BIBLIOGRAFÍA

1. Kong, X., et al., *A simplified binary harmony search algorithm for large scale 0–1 knapsack problems.* Expert Systems with Applications, 2015. **42**(12): p. 5337-5355.
2. Alejo-Machado, O., et al., *Fisherman search procedure.* Progress in Artificial Intelligence, 2014. **2**(4): p. 193-203.
3. Brotcorne, L., S. Hanafi, and R. Mansi, *A dynamic programming algorithm for the bilevel knapsack problem.* Operations Research Letters, 2009. **37**(3): p. 215-218.
4. Bhattacharjee, K.K. and S.P. Sarmah, *Shuffled frog leaping algorithm and its application to 0/1 knapsack problem.* Applied Soft Computing, 2014. **19**: p. 252-263.
5. Moosavian, N., *Soccer league competition algorithm for solving knapsack problems.* Swarm and Evolutionary Computation, 2015. **20**: p. 14-22.
6. Lim, T.Y., M.A. Al-Betar, and A.T. Khader, *Taming the 0/1 knapsack problem with monogamous pairs genetic algorithm.* Expert Systems with Applications, 2016. **54**: p. 241-250.
7. Wolpert, D.H. and W.G. Macready, *No free lunch theorems for optimization.* Trans. Evol. Comp, 1997. **1**(1): p. 67-82.
8. Leão, A.A.S., L.H. Cherri, and M.N. Arenales, *Determining the K-best solutions of knapsack problems.* Computers & Operations Research, 2014. **49**: p. 71-82.
9. Abecassis, F., et al., *Performance evaluation of CUDA programming for 5-axis machining multi-scale simulation.* Computers in Industry, 2015. **71**: p. 1-9.
10. Bukata, L., P. Šůcha, and Z. Hanzálek, *Solving the Resource Constrained Project Scheduling Problem using the parallel Tabu Search designed for the CUDA platform.* Journal of Parallel and Distributed Computing, 2015. **77**: p. 58-68.
11. Piamba, A.E.M., *Algoritmo para Generación Automática de Resúmenes Extractivos Genéricos de un Documento basado en el Procedimiento de Búsqueda del Pescador*. 2016, Universidad del Cauca.
12. Fukunaga, A.S., *A branch-and-bound algorithm for hard multiple knapsack problems.* Annals of Operations Research, 2009. **184**(1): p. 97-119.
13. Zou, D., et al., *Novel global harmony search algorithm for unconstrained problems.* Neurocomputing, 2010. **73**(16–18): p. 3308-3318.
14. Wang, L., et al., *An improved adaptive binary Harmony Search algorithm.* Information Sciences, 2013. **232**: p. 58-87.
15. Wang, L., et al., *An adaptive fuzzy controller based on harmony search and its application to power plant control.* International Journal of Electrical Power & Energy Systems, 2013. **53**: p. 272-278.

16. Xiang, W.-l., et al., *A Novel Discrete Global-Best Harmony Search Algorithm for Solving 0-1 Knapsack Problems.* Discrete Dynamics in Nature and Society, 2014. **2014**: p. 1-12.
17. Hanxiao, S. *Solution to 0/1 Knapsack Problem Based on Improved Ant Colony Algorithm*. in *Information Acquisition, 2006 IEEE International Conference on*. 2006.
18. Wang, Y., et al., *A novel quantum swarm evolutionary algorithm and its applications.* Neurocomputing, 2007. **70**(4–6): p. 633-640.
19. Li, Z. and N. Li. *A novel multi-mutation binary particle swarm optimization for 0/1 knapsack problem*. in *2009 Chinese Control and Decision Conference, CCDC 2009*. 2009.
20. Moosavian, N. and B. Kasaee Roodsari, *Soccer league competition algorithm: A novel meta-heuristic algorithm for optimal design of water distribution networks.* Swarm and Evolutionary Computation, 2014. **17**: p. 14-24.
21. Moosavian, N. and B.K. Roodsari, *Soccer League Competition Algorithm, a New Method for Solving Systems of Nonlinear Equations.* International Journal of Intelligence Science, 2014. **04**(01): p. 7-16.
22. Lin, F.-T., *Solving the knapsack problem with imprecise weight coefficients using genetic algorithms.* European Journal of Operational Research, 2008. **185**(1): p. 133-145.
23. Jiangfei, Z., et al. *Genetic Algorithm Based on Greedy Strategy in the 0-1 Knapsack Problem*. in *Genetic and Evolutionary Computing, 2009. WGEC '09. 3rd International Conference on*. 2009.
24. Yan, L. and L. Chao. *A Schema-Guiding Evolutionary Algorithm for 0-1 Knapsack Problem*. in *Computer Science and Information Technology - Spring Conference, 2009. IACSITSC '09. International Association of*. 2009.
25. Geem, Z.W., J.H. Kim, and G.V. Loganathan, *A new heuristic optimization algorithm: Harmony search.* Simulation, 2001. **76**(2): p. 60-68.
26. Eusuff, M. and K. Lansey, *Optimization of Water Distribution Network Design Using the Shuffled Frog Leaping Algorithm.* Journal of Water Resources Planning and Management, 2003. **129**(3): p. 210–225.
27. Eusuff, M., K. Lansey, and F. Pasha, *Shuffled frog-leaping algorithm: a memetic meta-heuristic for discrete optimization.* Engineering Optimization, 2006. **38**(2): p. 129-154.
28. Rahimi-Vahed, A., et al., *A novel hybrid multi-objective shuffled frog-leaping algorithm for a bi-criteria permutation flow shop scheduling problem.* The International Journal of Advanced Manufacturing Technology, 2008. **41**(11-12): p. 1227-1239.
29. Pan, Q.-K., et al., *An effective shuffled frog-leaping algorithm for lot-streaming flow shop scheduling problem.* The International Journal of Advanced Manufacturing Technology, 2010. **52**(5-8): p. 699-713.

30. Antunes, C.L., et al., *Parameter identification of Jiles-Atherton model using SFLA.* COMPEL - The international journal for computation and mathematics in electrical and electronic engineering, 2012. **31**(4): p. 1293-1309.
31. Fang, C. and L. Wang, *An effective shuffled frog-leaping algorithm for resource-constrained project scheduling problem.* Computers & Operations Research, 2012. **39**(5): p. 890-901.
32. Gomez-Gonzalez, M., F. Jurado, and I. Pérez, *Shuffled frog-leaping algorithm for parameter estimation of a double-cage asynchronous machine.* Electric Power Applications, IET. **6**(8): p. 484-490.
33. Roy, P., P. Roy, and A. Chakrabarti, *Modified shuffled frog leaping algorithm with genetic algorithm crossover for solving economic load dispatch problem with valve-point effect.* Applied Soft Computing, 2013. **13**(11): p. 4244-4252.
34. Gomez-Gonzalez, M., F.J. Ruiz-Rodriguez, and F. Jurado, *Probabilistic optimal allocation of biomass fueled gas engine in unbalanced radial systems with metaheuristic techniques.* Electric Power Systems Research, 2014. **108**: p. 35-42.
35. Luo, J. and M.-R. Chen, *Improved Shuffled Frog Leaping Algorithm and its multi-phase model for multi-depot vehicle routing problem.* Expert Systems with Applications, 2014. **41**(5): p. 2535-2545.
36. Pospichal, P., J. Schwarz, and J. Jaros. *Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu*. in *16th International Conference on Soft Computing MENDEL*. 2010.
37. Sajedi, H. and S.F. Razavi, *DGSA: discrete gravitational search algorithm for solving knapsack problem.* Operational Research, 2016.
38. Lv, J., et al., *Solving 0-1 knapsack problem by greedy degree and expectation efficiency.* Applied Soft Computing, 2016. **41**: p. 94-103.
39. Cobos, C., et al., *A Binary Fisherman Search Procedure for the 0/1 Knapsack Problem*, in *Advances in Artificial Intelligence: 17th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2016, Salamanca, Spain, September 14-16, 2016. Proceedings*, O. Luaces, et al., Editors. 2016, Springer International Publishing: Cham. p. 447-457.
40. Eshelman, L.J., *The CHC adaptive search algorithm: How to have safe search when engaging.* Foundations of Genetic Algorithms 1991 (FOGA 1), 2014. **1**: p. 265.
41. NVIDIA, *CUDA C Programming Guide PG-02829-001_v7.5*. 2015.
42. Nashed, Y.S.G., et al., *libCudaOptimize: an open source library of GPU-based metaheuristics*, in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. 2012, ACM: Philadelphia, Pennsylvania, USA. p. 117-124.
43. Pranav Bhandari, R.C., and Peter Yoon, *Shuffled Frog Leaping Algorithm for 0/1 Knapsack Problem on the GPU*, in *International Conference on Scientific Computing*. 2015.

44. Zou, D., et al., *Solving 0–1 knapsack problem by a novel global harmony search algorithm.* Applied Soft Computing, 2011. **11**(2): p. 1556-1564.
45. Jung, D., et al., *A New Parallelization Scheme for Harmony Search Algorithm*, in *Harmony Search Algorithm: Proceedings of the 2nd International Conference on Harmony Search Algorithm (ICHSA2015)*, H.J. Kim and W.Z. Geem, Editors. 2016, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 147-152.
46. We, J., et al. *A parallel harmony search algorithm with dynamic harmony-memory size*. in *2013 25th Chinese Control and Decision Conference (CCDC)*. 2013.
47. Le, K.C. and R. Abdullah. *Parallel Strategies for Harmony Search on CUDA to Solve Traveling Salesman Problem*. in *IT Convergence and Security (ICITCS), 2015 5th International Conference on*. 2015.
48. Pisinger, D., *Where are the hard knapsack problems?* Computers & Operations Research, 2005. **32**(9): p. 2271-2284.
49. Birattari, M., *Tuning Metaheuristics*, in *A Machine Learning Perspective*. 2009, Springer-Verlag Berlin Heidelberg. p. X, 221.
50. Jimena Adriana Timaná Peña, Carlos Alberto Cobos Lozada, and J.T. Jimenez, *Metaheuristic algorithms for building Covering Arrays: A review.* Revista Facultad de Ingeniería, 2016. **25**.
51. Lastra, M., D. Molina, and J.M. Benítez, *A high performance memetic algorithm for extremely high-dimensional problems.* Information Sciences, 2015. **293**: p. 35-58.
52. Luke, S., *Essentials of Metaheuristics*. 2013, Lulu
53. Mussi, L., F. Daolio, and S. Cagnoni, *Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture.* Information Sciences, 2011. **181**(20): p. 4642-4657.
54. NVIDIA, *POPULAR GPU-ACCELERATED APPLICATIONS CATALOG*. 2016.

Anexo A (Digital) Framework PMG_01KP

Anexo B (Digital) Archivos Excel con los resultados completos del ajuste de parámetros de las metaheurísticas paralelas usando la GPU, experimentos de las metaheurísticas secuenciales y experimentos de la metaheurísticas paralelas usando la GPU

Anexo C (Digital) Artículo "A Binary Fisherman Search Procedure for the 0/1 Knapsack Problem"

Anexo D (Digital) Artículo "Procedimiento de Búsqueda del Pescador Binario Paralelo usando la GPU para resolver el problema 0/1 knapsack para grandes dimensiones"