UNIVERSIDAD DE ALMERÍA

BISECCIÓN ITERATIVA DE UN SÍMPLEX REGULAR: PARALELIZACIÓN

Tesis doctoral presentada por
GUILLERMO APARICIO DE LAS LLANDERAS

Dirigida por
Dr. LEOCADIO GONZÁLEZ CASADO
Dr. ELIGIUS M. T. HENDRIX

UNIVERSIDAD DE ALMERÍA

BISECCIÓN ITERATIVA DE UN SÍMPLEX REGULAR: PARALELIZACIÓN

Tesis doctoral presentada por
GUILLERMO APARICIO DE LAS LLANDERAS

Dirigida por
Dr. LEOCADIO GONZÁLEZ CASADO
Dr. ELIGIUS M. T. HENDRIX

Almería, Noviembre de 2015

Bisección iterativa de un símplex regular: Paralelización.
Doctorando: Guillermo Aparicio de las Llanderas
Directores: Leocadio González Casado
Eligius M. T. Hendrix

La siguiente página web contiene información actualizada sobre los temas relacionados con esta tesis doctoral:
http://www.hpca.ual.es/
Texto impreso en Almería, Noviembre de 2015

A mi familia.

Prólogo

A menudo nos encontramos con situaciones en las que se desea conseguir el máximo o el mínimo de un determinado valor, pero siguiendo determinadas reglas. Por ejemplo, en una fábrica de muebles desean cortar los tablones de forma que se desperdicie el mínimo posible de madera, pero asegurando que las piezas creadas a partir de los tablones cumplen ciertas condiciones de calidad, forma etc. En general, los problemas de Optimización Global definen situaciones en las que se busca un máximo o un mínimo de una función objetivo de acuerdo a diferentes criterios. La aplicación práctica de la resolución de este tipo de problemas abarca campos de estudio que van desde la propia Matemática a la Ingeniería pasando por la Física, Biología o Economía, etc. Existen multitud de métodos de resolución de problemas de Optimización Global. En esta tesis nos vamos a centrar en el uso del método de Ramificación y Acotación, para un espacio de búsqueda definido por un símplex. Resolver este tipo de problemas cuando la dimensión y/o la precisión requerida son altas tiene un coste computacional muy alto, debido a esto, se hace necesario el uso de la computación paralela de altas prestaciones. Para obtener una mayor eficiencia en los algoritmos de Ramificación y Acotación paralelos, se requiere la determinación de la carga de trabajo de cada subproblema. De esta forma se puede distribuir esa carga de manera equitativa entre los diferentes elementos de proceso. Los árboles de búsqueda generados por los algoritmos de Ramificación y Acotación son unas estructuras de datos irregulares. Por lo tanto, es necesario el uso de un balanceo dinámico de la carga para obtener algoritmos paralelos eficientes. En esta tesis se va a utilizar el método de bisección de un símplex por su lado mayor como modo de refinamiento del símplex. Durante el proceso de bisección iterativa de un símplex regular se genera un árbol binario. Estamos interesados en conocer el tamaño de un subárbol a partir de un nodo cualquiera del árbol, para así poder realizar una estimación de la carga computacional pendiente.

Nuestro propósito es hallar los valores exactos del tamaño del árbol binario y no cotas superiores de su tamaño. El problema que aparece es que, para símplices de dimensión mayor que tres, la selección del lado mayor no es única, es decir, aparecen varios lados mayores que pueden ser divididos. Tradicionalmente, por

comodidad, se ha seleccionado, como lado a dividir, el primer lado mayor según la ordenación de los vértices del símplex.

Esta tesis se ha organizado de la siguiente forma: en el capítulo 1 se introduce el concepto de Optimización Global y se realiza una clasificación de este tipo de problemas. También se describen los algoritmos de Ramificación y Acotación que a menudo son utilizados para la resolución de problemas de Optimización Global. En esta tesis nos centraremos en problemas cuyo espacio de búsqueda es un símplex regular. En el capítulo 1 se presenta la definición de símplex y del método de bisección del lado mayor como modo de refinamiento del símplex regular. Este capítulo finaliza planteando las cuestiones a las que se dará respuesta en esta tesis.

En el capítulo 2 se da respuesta a la pregunta: ¿existe una heurística de selección del lado mayor a dividir que genere un árbol menor que la del primer lado mayor? Se estudian distintas heurísticas de selección del lado mayor que se han diseñado con la intención de reducir el número de símplices que aparecen en el árbol binario.

Para responder a la pregunta: ¿alguna de esas heurísticas obtiene el árbol de menor tamaño?, en el capítulo 3 se han desarrollado algoritmos secuenciales para responder a la siguiente pregunta: ¿cuál es el tamaño del menor árbol binario generado mediante bisección del lado mayor de un símplex regular?

Encontrar el menor tamaño del árbol es un problema de Optimización Combinatoria cuyo coste computacional crece exponencialmente con la dimensión y la precisión requeridas. En el capítulo 4 se plantea la siguiente pregunta: ¿cuál es el modelo de programación paralelo más eficiente para resolver este problema en sistemas multicore de memoria compartida? Para responder esta pregunta se estudian dos tipos diferentes de algoritmos paralelos, los que utilizan un número fijo de hebras durante su ejecución y los que hacen una gestión dinámica del número de hebras para resolver este tipo de problemas con altos requerimientos de memoria.

Para hacer una estimación del tamaño de un subárbol a partir de un nodo, el tener un árbol con el menor número de formas de símplices reduciría el número de

posibles subárboles y por lo tanto, facilitaría el cálculo de una estimación de su tamaño. En el capítulo 5 se presentan los conceptos forma de un símplex, semejanza de forma y clases de símplices. Además, se va a dar respuesta a la pregunta: ¿cuál es el menor número de formas de los subsímplices en el refinamiento de un símplex regular mediante bisección del lado mayor?

En el refinamiento de un símplex mediante bisección del lado mayor, hay que realizar repetidamente el cálculo de distancias para conocer los lados mayores susceptibles de ser divididos. Para reducir el coste computacional de este tipo de algoritmos se puede proporcionar una secuencia de lados mayores a dividir que genere el menor árbol binario. En el capítulo 6 se da respuesta a la cuestión: ¿cuál es la secuencia recurrente de lados mayores a dividir y de menor tamaño para obtener el árbol binario de menor tamaño?

El último capítulo de la tesis está dedicado a las conclusiones obtenidas del trabajo realizado y a presentar las futuras líneas de investigación.

Agradecimientos

En primer lugar me gustaría dar las gracias a Leocadio Gozález Casado por su ayuda. Nada de esto hubiese sido posible sin su confianza desde un primer momento. Gracias por su apoyo en el trabajo realizado y por la paciencia que ha demostrado tener para intentar convertir a un ingeniero en un científico.

Mi mas sincero agradecimiento a Eligius M.T. Hendrix e Inmaculada García Fernández por contribuir en el desarrollo de esta tesis de manera constructiva en todo momento.

A Boglárka Gazdag-Tóth que junto con Eligius han sido mi apoyo en cuestiones matemáticas durante estos años.

A Rafael Asenjo Plaza, que después de tantos años sigue enseñándome cosas nuevas cada vez que hablo con él.

A todos y cada uno de los miembros del grupo de investigación TIC-146, Supercomputación: Algoritmos. En especial a José Manuel García Salmerón por su ayuda y colaboración en multitud de ocasiones.

Al Plan nacional de I+D+I, por el contrato asociado al proyecto TIN2012-37483.

A mi familia, sin su apoyo, cariño y comprensión este trabajo no habría sido posible.

Noviembre de 2015

Acrónimos

B&B Branch and Bound. Algoritmo de Ramificación y Acotación.

BLM Bisección del Lado Mayor.

DNT-I-STS Dynamic Number of Threads Iterative Smallest Tree Size. Algoritmo iterativo paralelo de obtención del número de símplices del árbol binario de menor tamaño con un número dinámico de hebras.

DNT-R-STS Dynamic Number of Threads Recursive Smallest Tree Size. Algoritmo recursivo paralelo de obtención del número de símplices del árbol binario de menor tamaño con un número dinámico de hebras.

GT General Tree. Árbol que contiene todos los MABs que aparecen durante el proceso de refinamiento de un símplex.

I-STS Iterative Smallest Tree Size. Algoritmo iterativo de obtención del número de símplices del árbol binario de menor tamaño.

IPTH Iterative PThreads. Nombre abreviado de DNT-I-STS.

LE Longest Edge, análogo a LM.

LEB Longest Edge Bisection, análogo a BLM.

LLAlloc Lockless Allocator. Gestor de memoria dinámica de Lockless Inc®.

LM Lado Mayor.

MAB Menor Árbol Binario. Árbol binario con menor número de símplices que aparece durante el proceso de refinamiento de un símplex por su lado mayor.

NMAB-GT Número de Menores Árboles Binarios contenidos en un GT.

R-STS Recursive Smallest Tree Size. Algoritmo recursivo de obtención del número de símplices del árbol binario de menor tamaño.

RPTH Recursive Pthreads. Nombre abreviado de DNT-R-STS.

RS Refinamiento de un símplex.

S-LM Secuencia de Lados Mayores.

SQ Simplex Quality. Indicador de la calidad de un símplex.

T3S Tree 3 Steps. Nombre abreviado de T3S-R-STS.

T3S-R-STS Tree 3 Steps Recursive Smallest Tree Size. Algoritmo paralelo recursivo con un número fijo de hebras para la obtención del número de símplices del árbol binario de menor tamaño.

TBB Threading Building Blocks. Método de programación paralela de Intel® basado en conjuntos de tareas. Nombre abreviado de TBB-STS

TBB-STS Threading Building Blocks Smallest Tree Size. Algoritmo paralelo de obtención del número de símplices del árbol binario de menor tamaño basado en conjuntos de tareas.

TBBMalloc TBB Memory Alocator. Gestor de memoria dinámica de TBB.

TCMalloc Thread Caching Memory Allocator. Gestor de memoria dinámica de Google®.

TMAB Tamaño del Menor Árbol Binario. Número de símplices del menor árbol binario.

Índice general

Acrónimos ix

Índice de figuras xv

Índice de tablas xvii

1 Introducción 1

1.1 Optimización Global . . . 1

1.1.1 Definición del problema de Optimización Global . . . 2

1.1.2 Clasificación de los problemas de Optimización Global . . . 4

1.1.3 Clasificación de los algoritmos de resolución de problemas de Optimización Global . . . 5

1.2 Algoritmos Ramificación y Acotación . . . 8

1.3 Símplices: definición . . . 14

1.4 Ventajas del refinamiento mediante símplices . . . 16

1.5 Bisección de un símplex por su lado mayor . . . 18

1.6 Preguntas que surgen en esta investigación . . . 20

1.7 Conclusiones . . . 24

2 Heurísticas de selección del lado mayor a dividir 25

2.1 Descripción de las heurísticas . . . 25

2.1.1 Primer Lado Mayor, LEB_1 . . . 25

2.1.2 Lado mayor con menores ángulos en sus vértices, LEB_α . . . 26

2.1.3 Lado mayor con su punto medio lo más alejado posible del centroide del símplex, LEB_C . . . 26

2.1.4 Lado mayor con vértices que hayan intervenido menos en procesos de bisección, LEB_W . . . 26

2.1.5 Combinación de LEB_W con un segundo criterio , LEB_{W_2} . . . 27

2.1.6 Lado mayor con el punto medio más alejado del resto de los vértices, LEB_M . . . 27

2.2 Resultados 28
2.3 Conclusiones 29

3 Menor tamaño del árbol: alg. secuenciales 31
3.1 Algoritmo R-STS (Recursive - Smallest Tree Size) 33
3.2 Algoritmo I-STS (Iterative - Smallest Tree Size) 34
3.3 Estrategias para reducir la búsqueda 37
3.4 Resultados 38
3.5 Conclusiones 41

4 Menor tamaño del árbol: alg. paralelos 43
4.1 Introducción a la computación paralela 43
4.2 Versiones paralelas que utilizan un número fijo de hebras 51
4.2.1 Algoritmo Basado en Tres Fases 51
4.2.2 Algoritmo Intel Threading Building Blocks 56
4.3 Versiones paralelas que utilizan un número dinámico de hebras 60
4.3.1 Algoritmo paralelo con un número dinámico de hebras que realizan un procedimiento recursivo 61
4.3.2 Algoritmo paralelo con un número dinámico de hebras que realizan un procedimiento iterativo 63
4.4 Resultados 66
4.4.1 Resultados para las instancias I3 e I4 en BullX 67
4.4.2 Resultados para las instancias I3 e I4 en Aloe 68
4.4.3 Resultados para las instancias I5 e I6 69
4.5 Conclusiones 71

5 Alg. para determinar el árbol con el menor nº de clases 73
5.1 Clases de forma y medidas de calidad de un símplex 73
5.2 Medida de semejanza de forma entre dos símplices 77
5.3 Algoritmo para determinar el árbol con el menor número de clases . . 81
5.4 Resultados 83
5.5 Conclusiones 85

6 Menor secuencia de LMs que genera el menor árbol **87**
6.1 Condición m_k-válido 87
6.2 Algoritmo para generar todos los menores árboles binarios 90
6.3 Algoritmo para contar el número de menores árboles binarios 92
6.4 Algoritmo para determinar los árboles m_k-válidos 92
6.5 Resultados 94
6.6 Conclusiones 97

7 Conclusiones y trabajo futuro **99**
7.1 Conclusiones 99
7.2 Trabajo futuro 101

A Publicaciones que han contribuido a esta tesis **103**
A.1 Publicaciones en revistas internacionales 103
A.2 Publicaciones en proceedings de conferencias internacionales con DOI 104
A.3 Publicaciones en proceedings de otras conferencias internacionales . . 104
A.4 Publicaciones en proceedings de conferencias nacionales 105

Bibliografía **107**

Índice de figuras

1.1 Ejemplo de un problema con múltiples mínimos locales. 3
1.2 Símplices de dimensiones 2, 3 y 4. 15
1.3 2-símplex regular con una longitud de lado mayor igual a 1. 16
1.4 División de un hipercubo en símplices. 17
1.5 División de un 2-símplex por su lado mayor. 18
1.6 Bisección por el lado mayor de un 3-símplex regular. 19
1.7 Árbol binario generado por el algoritmo RS en un 2-símplex con $\epsilon = \frac{1}{2}$. 21
1.8 3-Símplices generados dependiendo del lado mayor seleccionado. . . 22

3.1 Estructura de datos de tipo Nodo 35

4.1 Ejemplo del algoritmo 4.1 para un 3-símplex cuando $NThread$=4 y $MinNLN$=8 53
4.2 Speedup para las instancias I3 e I4 en la máquina BullX. 68
4.3 Speedup para las instancias I3 e I4 en la máquina Aloe. 70
4.4 Speedup para la instancia I5 en las máquinas Bullx y Aloe. 71
4.5 Speedup para la instancia I6 en las máquinas Bullx y Aloe. 72

5.1 Obtención del ángulo α entre dos lados que comparten un vértice. . . 76
5.2 Generación de nuevas hojas en el árbol de árboles (ToT) cuando existen varios lados mayores de S_2. X_C no se va a analizar porque X_B tiene las mismas formas en los símplices descendientes que S_2. 84
5.3 Árbol binario generado por algoritmo RS (véase el algoritmo 1.2) para un 3-símplex usando en SelectLE() la tabla 5.1. 86

6.1 GT para un 3-símplex. Solo se evalúa un lado de S_1 porque es un símplex regular. S_2 se genera pero no se evalúa porque es simétrico a S_3. Lo mismo ocurre con los símplices S_6 y S_7. 91

Índice de tablas

1.1 Clasificación de los problemas de Optimización Global 4

2.1 Número de símplices del árbol binario obtenido por refinamiento de un símplex regular de tamaño unidad utilizando diferentes heurísticas. 28

3.1 Tiempo de ejecución del algoritmo 3.1 en segundos para las instancias I3 e I4 utilizando diferentes librerías de gestión de memoria dinámica en la máquina BullX. 40

3.2 Tiempo de ejecución de los algoritmos 3.1 y 3.4 en segundos para las instancias I3, I4, I5 e I6 en la máquina BullX. 41

3.3 Tiempo de ejecución de los algoritmos 3.1 y 3.4 en segundos para las instancias I3, I4, I5 e I6 en la máquina Aloe. 41

4.1 Estadísticas del algoritmo 4.1 para un 3-símplex cuando $\epsilon = 0,025$ y $MinNLN = NThreads$ 55

4.2 Estadísticas del algoritmo 4.1 para un 4-símplex cuando $\epsilon = 0,125$ y $MinNLN = NThreads$ 56

4.3 Estadísticas del algoritmo 4.1 para un 3-símplex cuando $\epsilon = 0{,}025$ y con las mejoras 4.1 y 4.2 57

4.4 Estadísticas del algoritmo 4.1 para un 4-símplex cuando $\epsilon = 0{,}125$ y con las mejoras 4.1 y 4.2 58

4.5 Resultados en Bullx para la instancia I3. $t_{sec} = 30{,}36$ s. 67

4.6 Resultados en Bullx para la instancia I4. $t_{sec} = 152{,}71$ s. 67

4.7 DMIPS y GB/s en BullX y Aloe además de la ganancia en DMIPS y GB/s de Aloe respecto a Bullx. 68

4.8 Resultados en Aloe para la instancia I3. $t_{sec} = 11{,}07$ s. 69

4.9 Resultados en Aloe para la instancia I4. $t_{sec} = 57{,}75$ s. 69

4.10 Resultados en Bullx para la instancia I5. $t_{sec} = 12.301{,}92$ s. 70

4.11 Resultados en Aloe para la instancia I5. $t_{sec} = 4.653{,}16$ s. 71

4.12 Resultados en BullX y Aloe para la instancia I6. $t_{sec-Bullx} = 15.566{,}24$ s. $t_{sec-Aloe} = 7.056{,}76$ s. 72

5.1 Secuencia de LMs a usar en SelectLE(S_i) del algoritmo 1.2, que ha sido obtenida mediante el algoritmo 5.1 para obtener el árbol con el menor número de clases en un 3-símplex. 85

6.1 A-3SNC es un ejemplo de conversión a matriz de la tabla 5.1. $A_{l\times 2}$ puede extenderse a $A_{l\times 4}$. 94

6.2 Tamaño del menor árbol binario (TamañoArbol()) y del árbol binario generado por RS() con la matriz A-3SNC para un 3-símplex, variando la precisión ϵ. 95

6.3 Resultados de los algoritmos 6.1, 6.2 y 6.3 para un 3-símplex y $k = 2$. 96

Listado de Algoritmos

1.1 RyA(S_1, ϵ) . . . 14
1.2 RS(S_1, ϵ) . . . 20
1.3 Bisect(S_i, j, k) . . . 20
3.1 R-STS (S_i, ϵ) . . . 34
3.2 CreaNodo (S_i, PN, p) . . . 36
3.3 CreaNodosHijo (N) . . . 36
3.4 I-STS (S_1, ϵ) . . . 37
4.1 3S-R-STS(S_1, ϵ, $NThreads$, $MinNLN$) . . . 54
4.2 Thread-3S-R-STS(Λ, ϵ) . . . 54
4.3 Gather-3S-R-STS(N) . . . 55
4.4 TBB-STS(S_1, ϵ, $NThreads$) . . . 58
4.5 TBB-R-STS (S_i, ϵ) . . . 59
4.6 DNT-STS (S_1, ϵ, $NThreads$, ***Método)*** . . . 61
4.7 DNT-R-STS (S_i, ϵ) . . . 62
4.8 DNT-I-STS (N, ϵ) . . . 64
4.9 PróximoHijoAProcesar(N, Rem, $Finish$) . . . 65
5.1 MinClassB&B(S_1) . . . 83
6.1 TMAB (S, ϵ) . . . 90
6.2 NMAB-GT (N) . . . 92
6.3 m_k-MAB (GT, k) . . . 93

Capítulo

1

Introducción

En este capítulo se van a presentar conceptos que van a ser desarrollados durante esta tesis. En primer lugar se va a realizar una introducción a los problemas de Optimización Global. Tras ello, se van describir los algoritmos de Ramificación y Acotación, también conocidos como Branch and Bound (B&B), que se usarán para resolver este tipo de problemas. En este capítulo se introduce el concepto de símplex y su uso como espacio de búsqueda en los algoritmos de Ramificación y Acotación. Se propone la bisección de un símplex por su lado mayor como modo de refinamiento. Este capítulo finaliza presentando las cuestiones que se pretenden investigar a lo largo de la tesis.

1.1 Optimización Global

La optimización trata de dar respuesta a la pregunta "¿qué es mejor?" para problemas donde la calidad de la respuesta puede expresarse mediante un valor. Se trata de buscar un máximo o un mínimo de una función f de acuerdo a diferentes criterios. Los primeros problemas de optimización se resolvieron en la antigua Grecia y fueron considerados como uno de los descubrimientos más significativos de la época. En el siglo primero d.C., el matemático alejandrino Herón resolvió el problema de encontrar el camino más corto entre dos puntos cuando se ha de pasar por un camino intermedio utilizando la técnica del espejo. Básicamente lo que deseaba saber era el punto en el que debería recoger agua de un río cuando debía ir de una ciudad a otra, de modo que el camino recorrido fuese mínimo. La solución a este problema se conoce como el teorema de Herón y se considera como el origen de la geometría óptica actual. El problema de encontrar valores extremos ganó especial importancia en el siglo XVII cuando sirvió como una de las motivaciones en la invención del cálculo diferencial, que es la base

de la teoría moderna de optimización matemática. La invención de las computadoras ha provocado el avance en el campo de la optimización numérica. Durante la segunda guerra mundial, los algoritmos de optimización se utilizaron para resolver problemas militares de logística y operaciones. La optimización se ha convertido en uno de los campos más importantes y más estudiados en la matemática aplicada y la ingeniería desde entonces.

Un algoritmo de optimización puede hallar múltiples soluciones posibles dentro de los parámetros del problema. Estas soluciones tratan de maximizar o minimizar una función objetivo y se llaman soluciones locales. A la mejor de todas ellas la llamaremos solución global. La búsqueda de esta solución global se realiza en mediante los algoritmos de Optimización Global. Además de las matemáticas, la aplicación práctica de la resolución de problemas de Optimización Global abarca campos como la Economía, Física, Química, Biología o Ingeniería. Por ejemplo en la industria aparece el problema de optimizar el corte de las placas de material de modo que la parte sobrante que se desperdicia sea la mínima. En compañías de negocios, un ejemplo claro es como distribuir los recursos disponibles de modo que se optimice el beneficio.

En este capítulo se va a proporcionar una descripción básica de los conceptos que serán utilizados a lo largo de la tesis para mejorar algunos de los algoritmos usados en la resolución de problemas de Optimización Global.

1.1.1 Definición del problema de Optimización Global

Ya que un problema de maximización puede reescribirse como uno de minimización:

$$\max_{x \in S}\{f(x)\} = \min_{x \in S}\{-f(x)\} \tag{1.1}$$

un problema de Optimización Global se puede describir como:

$$\begin{aligned} minimizar\ & f(x) \\ sujeto\ a:\ & g(x) \leq 0, \\ & h(x) = 0, \\ & \underline{x} \leq x \leq \overline{x}, \\ & x \in \mathbb{R}^n \end{aligned} \tag{1.2}$$

donde $f(x) : \mathbb{R}^n \to \mathbb{R}$ es la función objetivo y $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ y $h(x) : \mathbb{R}^n \to \mathbb{R}^k$ son funciones que restringen la región de búsqueda de la solución. Se define la región de búsqueda de los posibles candidatos, o dominio del problema (1.2), como:

$$S = \{x \in \mathbb{D} \subseteq \mathbb{R}^n : g(x) \leq 0,\ h(x) = 0\} \tag{1.3}$$

donde $\mathbb{D}$ es el espacio de búsqueda.

Una solución local $(\tilde{x}, \tilde{f})$ del problema (1.2), se define como $\tilde{f} = f(\tilde{x}) \leq f(x), \forall x \in S \cap N_\epsilon(\tilde{x})$ con $\epsilon > 0$, donde $N_\epsilon(\tilde{x})$ son los ϵ-vecinos del punto $\tilde{x}$, definidos por $N_\epsilon(\tilde{x}) = \{x : \parallel x - \tilde{x} \parallel < \epsilon\}$. Para que esta solución local sea solución global $x^* \in S$ ha de cumplir que $f^* = f(x^*) \leq f(x),\ \forall x \in S$.

Un mínimo local no se considera una solución óptima para (1.2) hasta que no se demuestra que es una solución global. Existe una amplia variedad de problemas donde la existencia de un solo mínimo no puede ser postulada o verificada, con lo que su resolución se vuelve más difícil. Ejemplos de esos problemas pueden encontrarse en [1, 17, 18, 21, 54, 62, 73].

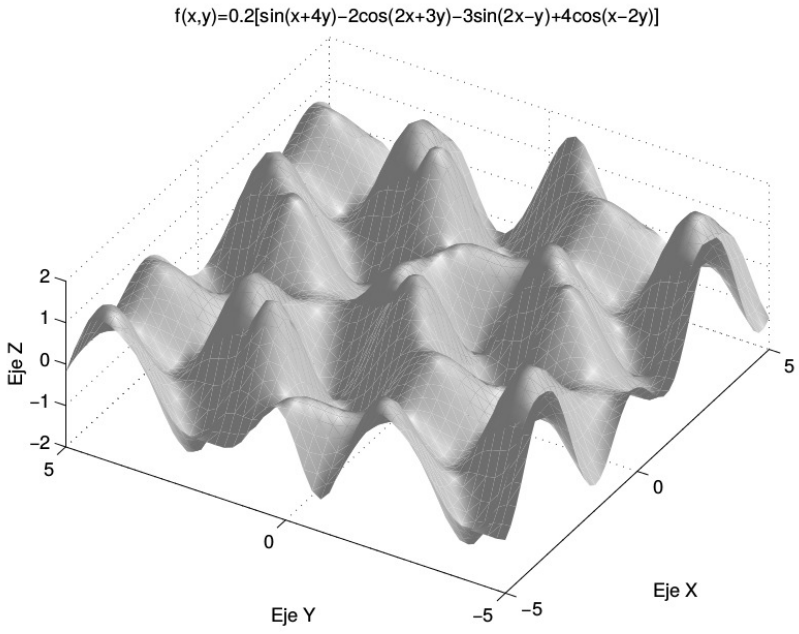


Figura 1.1: Ejemplo de un problema con múltiples mínimos locales.

A menudo existen multitud de mínimos locales, pero no todos ellos son globales. Podemos ver un ejemplo de ello en la figura 1.1 que muestra una composición de funciones trigonométricas con argumentos polinomiales, sobre un espacio de búsqueda bidimensional. La repercusión de no encontrar un mínimo global de manera correcta puede ser muy alta, no solo en términos científicos sino también económicos debido al amplio espectro de uso de este tipo de algoritmos. Debido a todo esto es muy importante utilizar una estrategia de búsqueda global correcta.

1.1.2 Clasificación de los problemas de Optimización Global

La tabla 1.1 muestra una clasificación general de los diferentes problemas de Optimización dependiendo del número de variables, de su tipo y del tipo de funciones involucradas en el problema de Optimización Global[1].

Tabla 1.1: Clasificación de los problemas de Optimización Global

Característica	**Propiedad**	**Clasificación**
Número de variables	Una	Unidimensional
	Varias	Multidimensional
Tipo de variables	Reales continuas	Continua
	Enteras	Discreta
	Reales continuas y enteras	Entera mezclada
	Permutaciones de enteros	Combinatoria
Tipo de Funciones	Funciones lineales	Lineal
	Funciones cuadráticas	Cuadrática
	Otras funciones no lineales	No Lineal
Formulación del problema	Sujeto a restricciones	Con restricciones
	No sujeto a restricciones	Sin restricciones

En esta tesis se van a tratar problemas de Optimización Global de tipo multidimensional. Cuando las variables de los problemas de Optimización son variables enteras, estos problemas se denominan también problemas de *Programación Entera* (IP, Integer Programming). Nosotros usaremos funciones continuas ya que si todas la funciones definidas en el problema son continuas, el conjunto de soluciones óptimas para el

[1] http://www.phy.ornl.gov/csep/mo/mo.html

problema (1.2) es no vacío. Si las funciones del problema son lineales, se llaman problemas de *Programación Lineal* (LP, Linear Programming). Si por el contrario, dichas funciones son no lineales, caso que nos ocupa, nos encontramos ante problemas de *Programación No Lineal* (NLP Non Linear Programming) Si bien la dificultad de los problemas de optimización lineal depende directamente del número de restricciones lineales, en los problemas de optimización no lineal, que son los que trataremos en esta tesis, la complejidad crece exponencialmente a medida que se añaden variables al problema.

Si se desea una clasificación más concreta de los problemas de Optimización Global se puede recurrir a [61], [36] y [30], donde existen referencias a diferentes problemas de optimización como: Programación Bilineal o Biconvexa, Optimización Combinatoria, Minimización Cóncava, Optimización Global Continua, Diferencial Convexa (DC), Programación Fraccional, Problemas Lineales y No Lineales Complementarios, Optimización Lipschitz, Problemas MiniMax, Optimización Multinivel, Programación Multiobjetivo, Programación Multiplicativa, Problemas de Redes de Trabajos, Programación Paramétrica No Convexa, Optimización Cuadrática, Programación Inversa Convexa, Optimización Global Separable y otros modelos probabilísticos no convexos.

Una vez que hemos definido el tipo de problema que deseamos resolver, podemos hacer un análisis previo de la complejidad esperada para este. En [55] Pardalos y Schnitger demuestran que hallar un posible óptimo local, para problemas de Programación Cuadrática, es un problema NP-Duro. En [56] se observa como la Programación Cuadrática cuando existe un autovalor negativo es un problema NP-Duro. Por lo tanto, la resolución del problema no convexo más sencillo de optimización No Lineal, es NP-Duro. Como se explica en [19], el tiempo de resolución de cualquier algoritmo NP-Duro, en el peor caso, se incrementa exponencialmente con el tamaño del problema y debido a ello, incluso problemas de tan solo dimensión diez, pueden ser considerados como “grandes”, dependiendo del problema de Optimización Global investigado.

1.1.3 Clasificación de los algoritmos de resolución de problemas de Optimización Global

A modo de resumen, las cualidades deseadas para un algoritmo de Optimización Global deben ser:

- **Corrección:** Producir resultados correctos.
- **Completitud:** Encontrar todas las posibles soluciones.
- **Finitud:** Garantizar la convergencia.
- **Certeza:** Probar la existencia o inexistencia de soluciones.

Si bien existen muchas clases de algoritmos de Optimización Global que se pueden implementar y aplicar directamente garantizando teóricamente su convergencia, tan solo unos pocos garantizan que se cumplan todas las cualidades mostradas.

Tal y como se ha visto, los problemas de Optimización Global pueden ser bastantes complejos de resolver incluso para las instancias más simples. Cuando se requiere una eficiencia mayor, los algoritmos de Optimización Global pueden utilizar fases de optimización local tradicionales que convergen rápidamente cuando están cerca de un óptimo local. Sin embargo, estos métodos tradicionales pueden fallar, no obteniendo una convergencia en problemas difíciles de resolver. Se desea que la convergencia global se garantice en todo momento, por lo que no solo es necesario desarrollar algoritmos eficientes de Optimización Global sino que también es necesario que estos sean robustos.

Se pueden hallar multitud de clasificaciones de los algoritmos de resolución de algoritmos de Optimización Global, pero una de las más utilizadas es la que divide las estrategias existentes en estrategias de búsqueda estocásticas y estrategias de búsqueda deterministas.

Métodos estocásticos: Los métodos estocásticos poseen un comportamiento no determinista, lo que significa que cada uno de los estados puede estar formado por elementos aleatorios y la forma de llegar a cada estado también puede ser aleatoria. No obstante, según [42, 52] cualquier desarrollo temporal que pueda ser analizable en términos de probabilidad puede ser denominado como un método estocástico. Todos los métodos estocásticos usan algún factor aleatorio en sus algoritmos y la demostración de su convergencia depende de argumentaciones estadísticas [71, 73]. Normalmente los métodos estocásticos se aplican a problemas sin restricciones y no necesitan ningún conocimiento sobre la función objetivo, dando a menudo resultados óptimos o cercanos a los óptimos. Se basan en obtener valores de la función objetivo en puntos de la región de búsqueda elegidos aleatoriamente. Estos métodos pueden requerir un número aleatorio

de iteraciones para alcanzar una solución al problema. Son el recurso a utilizar cuando el tamaño del problema, dimensionalmente hablando, es grande o el problema no está definido analíticamente. Las desventajas de estos métodos es que, al emplear un número no determinado de iteraciones, no garantizan que se encuentre la solución en un número finito de pasos, ni que el mínimo encontrado sea el global.

Métodos determinísticos: Al contrario que los métodos estocásticos, los métodos determinísticos no toman decisiones aleatorias en sus algoritmos y sus demostraciones de convergencia no dependen de la probabilidad. Es decir, son métodos que alcanzan siempre la misma solución para una entrada particular. Algunos ofrecen un tiempo de terminación finito para problemas específicos y otros convergen cuando el número de iteraciones se aproxima a infinito. Dentro de los algoritmos determinísticos se incluyen los métodos de Ramificación y Acotación, que se caracterizan por realizar un búsqueda exhaustiva del espacio de búsqueda y de encontrar todas las posibles soluciones hasta cierta precisión, ϵ.

La clasificación que se acaba de presentar es bastante clásica. Clasificaciones más actuales, como la presentada por Neumaier en [53], establecen clases de algoritmos de optimización en función de la rigurosidad de la solución obtenida por dichos algoritmos. Se diferencia entre:

Métodos incompletos. Aquellos que pueden quedar atrapados en un mínimo local.

Métodos asintóticamente completos. Aquellos que alcanzan siempre el mínimo global en una ejecución suficientemente larga, pero que no tienen información acerca de cuando un mínimo global es encontrado.

Métodos completos. Aquellos que alcanzan con certeza el mínimo global (en ausencia de errores en los cálculos) en una ejecución suficientemente larga y que además conocen una aproximación a la solución global durante su ejecución.

Métodos rigurosos. Aquellos que alcanzan con certeza el mínimo global para la precisión deseada incluso cuando existen errores en los cálculos.

1.2 Algoritmos Ramificación y Acotación

Los métodos que exploran todo el espacio de búsqueda se conocen como métodos completos o exhaustivos. Estos métodos aseguran la localización del mínimo global con la precisión deseada al finalizar el proceso. Los algoritmos de Ramificación y Acotación (RyA), también conocidos como Branch and Bound (B&B), son unos de los métodos exhaustivos más utilizados.

Los primeros algoritmos de Ramificación y Acotación aparecieron en artículos de los cincuenta y principios de los sesenta, donde los investigadores describieron esquemas enumerativos para resolver, lo que posteriormente se denominaron, problemas NP-Completos. Debido a la generalidad del método y la efectividad que aportaba, fue ampliamente usado. De hecho, todavía es una de las mejores herramientas para resolver problemas difíciles. Los primeros que dieron el nombre de Ramificación y Acotación a este método, fueron Little, Murty, Sweeny y Karel [47], en 1963, en su artículo sobre el problema del viajante de comercio. Lawer y Wood [46] estudiaron los algoritmos de Ramificación y Acotación, obteniendo una descripción independiente del problema, siendo así el primer artículo que presenta un modelo general de Ramificación y Acotación.

Los algoritmos de Ramificación y Acotación son ampliamente utilizados para la resolución de problemas de optimización combinatoria. Además se han convertido en una de las principales herramientas para resolver problemas de Optimización Global con variables continuas como pueden ser: minimización cóncava, minimización de funciones de Lipschitz [59, 60], optimización DC (donde el problema se puede convertir en la diferencia entre dos funciones convexas), obtención de regresiones de mínimos cuadrados no lineales [77], detección de copositividad [78], problemas de mezclas [3], etc.

La idea principal del método B&B consiste en subdividir el conjunto de posibles soluciones, al que se denomina espacio de búsqueda, en subconjuntos disjuntos de menor tamaño creando un árbol de búsqueda. Para cada subconjunto o nodo del árbol se determinan cotas superiores y/o inferiores del valor de la función objetivo. Los subconjuntos en los que las cotas empeoren los mejores valores que se han encontrado hasta el momento, son descartados en la búsqueda ya que no van a contener la solución global del problema. Los algoritmos de Ramificación y Acotación son uno de los

pocos métodos completos que resuelven problemas de Optimización Global.

El nodo más prometedor es el primero en visitarse y dividirse. Los criterios para elegir el nodo más prometedor se basan en cuestiones heurísticas y se conoce como estrategia de selección. Si en la estrategia de selección no influyen factores aleatorios, el algoritmo tendrá un comportamiento determinista.

Al final del proceso se obtiene una lista de nodos que determina la región o regiones donde se localiza el mínimo global de la función y cotas superiores e inferiores de su valor. Cuanto mayor sea la precisión deseada para la solución, menor será el tamaño de la región o regiones obtenidas en donde se encuentra el mínimo global y mayor será el tiempo de cómputo necesario para encontrarlas.

Una descripción de los modelos de algoritmos de Ramificación y Acotación que aparecen en los artículos [37, 39, 40, 45, 46, 50, 51], hecha por Bruin, Kindervater y Trienekens puede encontrarse en [10]. Estos autores no discuten modelos más recientes debido a que sólo se diferencian de los anteriores en pequeñas variaciones. Esta revisión de los algoritmos de Ramificación y Acotación desea establecer un esquema común para todas las aplicaciones, mediante la descripción de unas reglas u operadores generales [37, 50]. Para especificar estas reglas nos basaremos en los formalismos utilizados por Corrêa y Ferreira [9], centrándonos en problemas donde se intenta encontrar el valor mínimo de una función objetivo.

Los algoritmos B&B se basan en cinco *reglas básicas*. Las cuatro primeras son las que caracterizan este tipo de algoritmos según Mitten e Ibaraki [37, 50], pero además dependiendo del tipo de problema, es necesario añadir la quinta y última regla.

- *Regla de División*: Establece cómo se descomponen los nodos del árbol de búsqueda.
- *Regla de Acotación*: Calcula un *límite inferior* de la solución óptima de un nodo.
- *Regla de Selección*: Define qué nodo será el siguiente a dividir.
- *Regla de Eliminación*: Establece cómo reconocer y eliminar nodos donde no se encuentra la mejor solución del problema original.
- *Regla de Terminación*: Determina cuándo un nodo pertenece a la solución final. Esta regla aparece, por ejemplo, en problemas donde la solución del problema está determinada por la precisión deseada. Por otro lado, en algunos problemas

tradicionales de Optimización Combinatoria, el criterio de terminación es inherente a obtener una solución factible. Por ejemplo, en el problema del Viajante de Comercio [11, 47, 66, 76], donde hay que encontrar el camino mínimo para visitar todas las ciudades deseadas.

Los algoritmos de Ramificación y Acotación consisten en una secuencia de iteraciones en las que se aplican las reglas básicas a una estructura de datos Λ, que puede verse como un conjunto ordenado de subproblemas [7] y a los subproblemas que se extraen de esta estructura. Estas reglas seleccionan un subproblema de Λ, lo descomponen y eventualmente insertan los subproblemas creados en la estructura de datos Λ. A cada subproblema se le asocia una solución factible, resolviendo el subproblema si es suficientemente simple o asignándole una solución, elegida entre las posibles dentro del subproblema (no necesariamente la mejor). La mejor solución factible encontrada durante la ejecución del algoritmo de Ramificación y Acotación, será un límite superior de la solución final y se utilizará para eliminar aquellos subproblemas generados que no pueden contener una solución factible mejor que la que ya se ha encontrado [37, 40, 50, 75].

A continuación se describen las heurísticas utilizadas en la regla de selección:

Búsqueda en Profundidad (Depth-First)

Esta regla de selección selecciona, de entre aquellos subproblemas que estén a más profundidad en el árbol de búsqueda, el que tenga un menor valor de una función de prioridad heurística h. En este método se usa una estructura de almacenamiento LIFO (Last-In-First-Out) o pila. El árbol asociado a este tipo de selección es aquel en el que uno de los nodos hijo se expande hasta obtener una solución factible del problema o un subproblema no factible. La complejidad espacial de la pila, $O(lw)$, se incrementa linealmente con el nivel del árbol de búsqueda alcanzado, l, y el factor de división w [48]. Al descomponer el subproblema elegido, los subproblemas generados a partir de él se introducen en la estructura LIFO, siguiendo un orden decreciente de h.

Desventajas:

- El número de subproblemas inspeccionados es normalmente mayor que el realizado en otros tipos de selecciones, como la de *Primero el Mejor*.

- Si se entra en una rama del árbol de búsqueda que no lleva a la solución óptima, se necesita mucho tiempo para salir de esa ramificación.

Ventajas:

- Es el método de selección más económico desde el punto de vista del espacio de memoria requerido.
- Conduce más rápidamente a la obtención de una mejor cota superior del valor del mínimo global.

Búsqueda en Anchura (Breadth-First)

En la búsqueda en anchura, se elige, de entre aquellos subproblemas que estén a menor profundidad en el árbol de búsqueda, el que tenga un menor valor de una función de prioridad heurística h. Este método usa una estructura de almacenamiento FIFO (First-In-First-Out) o cola.

Desventajas:

- Este tipo de estrategia no es recomendable ni desde el punto de vista del tiempo de computación ni desde el del ahorro de memoria.

Ventajas:

- Admite la aplicación de un test de dominancia entre los subproblemas que se encuentran a la misma profundidad del árbol de búsqueda, para algunos tipos de problemas.

Primero el Mejor (Best-First)

La regla de selección elige aquel subproblema, con un menor valor de una función de prioridad heurística, independientemente del nivel de profundidad del árbol de búsqueda en el que se encuentre. La estructura más simple de almacenamiento es una lista ordenada.

Desventajas:

- La mejor cota superior del valor mínimo global, suele obtenerse en las fases finales del algoritmo.

- Aparece un crecimiento exponencial en el espacio de memoria necesario cuando se incrementa la profundidad del árbol de búsqueda. La complejidad espacial es $O(w^l)$.

Ventajas:

- Difícilmente un subproblema es descompuesto innecesariamente. En [20] se demuestra que usando *Best-First* se exploran menos nodos que usando otra estrategia, cuando hay que encontrar todas las soluciones óptimas, no siendo cierto cuando sólo se requiere encontrar una.

En general la función h está determinada por el límite inferior de la función objetivo en el subproblema, pero existen otras posibles opciones para h [41]. Después de la selección de un subproblema N, la *Regla de División* crea un conjunto de nuevos subproblemas a partir de N.

Definición 1.1. *La Regla de División de un nodo, en un algoritmo de Ramificación y Acotación corresponde a:*

- *Una partición del nodo, si N no es un nodo solución, es decir, no es lo suficientemente pequeño para ser resuelto.*
- *La solución del problema asociado al nodo, en otro caso.*

Sea $f^*(N_i)$ el valor óptimo de la solución del nodo N_i, que será descompuesto en los nodos $N_{i1}, N_{i2}, \ldots, N_{im}$ por la regla de división. Entonces, se tiene que:

$$f^*(N_i) = \min_{k=1,\ldots,m} f^*(N_{ik})$$

Es decir, el valor óptimo de la solución de un nodo se puede obtener mediante la evaluación de sus nodos hijos. En la práctica, las reglas de división cumplen que cada solución factible de un nodo padre es también solución de al menos uno de sus hijos. Estas reglas de división dependerán del problema concreto en el que se aplique el algoritmo de Ramificación y Acotación.

En cada iteración se puede generar un nuevo límite superior de la solución global, $\overline{f^*}$, de dos formas: porque el nodo generado por descomposición es un nodo solución, o porque se encuentra una solución factible mejor durante la computación de una partición en una descomposición.

La *Regla de Eliminación* permite hacer una búsqueda inteligente en S que evite considerar subproblemas que se sabe que no conducirán a la mejor solución del problema original.

Definición 1.2. *La Regla de Eliminación, en cada iteración, elimina todos los subproblemas cuya cota inferior sea mayor que el límite superior que se tiene de la solución global [8, 37].*

No existe una regla más importante que las demás, ya que todas ellas cooperan conjuntamente para la resolución del problema. El objetivo principal es reducir el tamaño del árbol de búsqueda para obtener rápidamente la mejor solución. Aunque la *Regla de Eliminación* es la que realiza la poda del árbol en última instancia, será aplicada en mayor o menor medida dependiendo de las demás reglas. Una *Regla de Acotación* que obtenga un buen límite inferior de la posible solución de un subproblema, permitirá caracterizar mejor a los subproblemas que no contienen la mejor solución, para eliminarlos. Para poder eliminar subproblemas, también hay que encontrar un buen límite superior de la solución. Este límite superior se consigue inspeccionando primero los nodos más prometedores, ya que son los que pueden aportar mejores soluciones. Este orden se establece mediante la *Regla de Selección*, de la que también dependen los requerimientos de memoria del problema. Hay que distinguir entre el número total de subproblemas inspeccionados y el número máximo de subproblemas almacenados en la estructura de datos con los nodos pendientes de evaluar. La gestión de esta estructura de datos ordenada será más costosa cuanto mayor sea el número de subproblemas que contenga. La *Regla de División* también juega un papel importante ya que de ella dependerá el número de ramas generadas a partir de un nodo del árbol de búsqueda. Por un lado, generar muchas ramas permite obtener información más precisa debido a que los subproblemas generados son menores en tamaño, pero por otro lado, hay que inspeccionar más nodos del árbol en cada división. No sólo es importante el nivel de división realizado, sino también cómo se realiza la división. Por lo tanto, la *Regla de División* debe estar orientada a reducir el espacio de búsqueda pero sin generar un número excesivo de nodos en cada división.

Cuando se requiere una búsqueda exhaustiva del mínimo global, se suelen utilizar algoritmos de Ramificación y Acotación debido a que las cualidades de esta clase de algoritmos son: Completo, encuentra todas las posibles soluciones; Finito, garantiza la

convergencia del algoritmo; y Certero, encuentra la solución si la hay. Por otra parte, no tiene la cualidad de Correcto, pues la aritmética computacional tiene errores de redondeo que puede producir soluciones no exactas, debido a esto, es necesario tener en cuenta siempre margenes de error en las operaciones aritméticas.

Un algoritmo de Ramificación y Acotación, como el algoritmo 1.1, realiza una búsqueda mediante la descomposición sucesiva del problema inicial en subproblemas de menor tamaño hasta alcanzar la(s) solución(es) final(es) o una aproximación a ella(s). Esta aproximación está determinada por la precisión requerida para la solución. El algoritmo 1.1 crea un árbol en el que una rama se poda cuando se determina que un subproblema no contiene una solución global. Para realizar la poda o rechazo de un subproblema se suelen calcular cotas de los valores de la función objetivo para cada subproblema.

Algoritmo 1.1 RyA(S_1, ϵ)

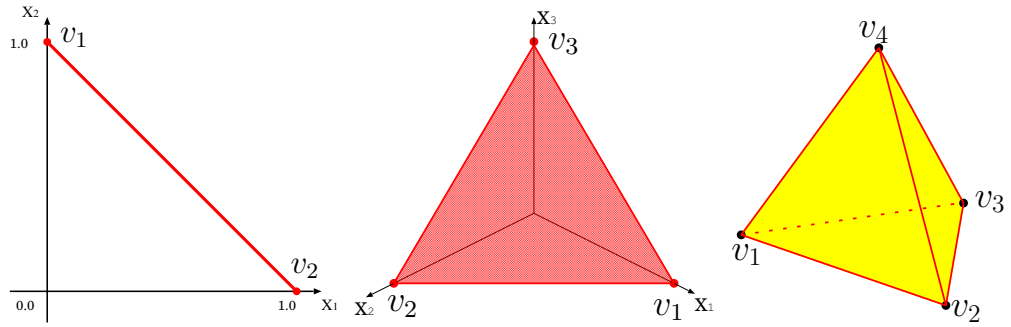
Requiere: S_1: región inicial, ϵ: precisión.

```
1:  Λ := {S_1}                                   ▶ Conjunto de trabajo
2:  Ω := {}                                      ▶ Conjunto final
3:  while Λ ≠ ∅ do
4:      Selecciona S_i de Λ                      ▶ Regla de selección
5:      Evalúa S_i                               ▶ Regla de acotación
6:      if S_i no puede ser eliminado then       ▶ Regla de rechazo
7:          if w(S_i) ≤ ε then                   ▶ Regla de terminación
8:              Almacena S_i in Ω
9:          else
10:             {S_1, .., S_j} := Divide(S_i)    ▶ Regla de división
11:             Almacena {S_1, .., S_j} en Λ
12: return Ω
```

1.3 Símplices: definición

Un símplex es el politopo más sencillo posible en un espacio dado, es decir, un n-símplex será el poliedro con el menor número de vértices posible en un espacio $n+1$ dimensional. Así, un 1-símplex es un segmento de una línea; un 2-símplex un triángulo; un 3-símplex es un tetraedro. Un n-símplex está definido en un espacio de $n+1$

dimensiones y tendrá $n + 1$ vértices. La figura 1.2 muestra gráficamente los ejemplos anteriores.



(a) 1-Símplex. (b) 2-Símplex. (c) 3-Símplex.

Figura 1.2: Símplices de dimensiones 2, 3 y 4.

Se puede definir un símplex del siguiente modo:

Definición 1.3. *Un símplex S es la envoltura convexa de un conjunto de n+1 puntos $v_1, \ldots, v_{n+1} \in \mathbb{R}^m$ afinmente independientes, en un espacio euclídeo de dimensión m mayor o igual que n ($m \geq n$), es decir, el conjunto de puntos tal que ningún k-plano contiene más de k+1 de estos puntos con $k \leq n$.*

$$S = \left\{ x = \sum_{j=1}^{n+1} \lambda_j v_j | \sum_{j=1}^{n+1} \lambda_j = 1, \lambda_j \geq 0, j = 1, \ldots, n+1 \right\} \tag{1.4}$$

Un n-símplex está definido por su conjunto de vértices $V = \{v_1, \ldots, v_{n+1}\}$, $v_j \in \mathbb{R}^m$. Cada vértice es un vector unidad. La figura 1.3 muestra un 2-símplex que ha sido escalado para que la longitud de lado sea igual a uno. Se ha utilizado la siguiente notación:

- $\omega(S)$: es el tamaño o ancho de un símplex S que es la longitud de su lado mayor, es decir, $\max_{i,j} \|v_i - v_j\|$.
- $|V|$: es el número de vértices.

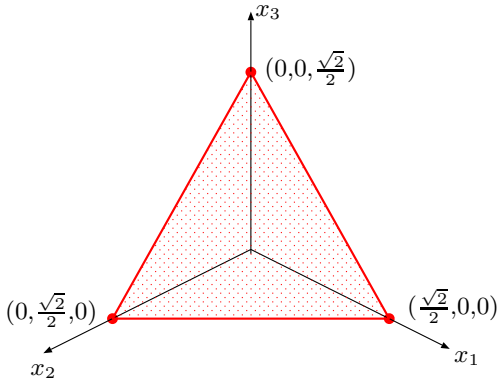


Figura 1.3: 2-símplex regular con una longitud de lado mayor igual a 1.

1.4 Ventajas del refinamiento mediante símplices

En un algoritmo de Ramificación y Acotación, las reglas de división y acotación dependen de la forma del espacio de búsqueda o región factible y del tipo de particionamiento utilizado. Los espacios de búsqueda en problemas de Optimización Global pueden ser hiper-rectangulares, basados en símplices, hiper-cónicos o hiper-esféricos. Dependiendo del problema a resolver, el espacio de búsqueda podrá ser de un tipo o de otro. El tipo de particionamiento utilizado también podrá variar, pudiendo incluso ser diferente para un mismo tipo de problema. Por ejemplo, cuando se resuelven problemas de optimización utilizando algoritmos B&B en los que el espacio de búsqueda es un hiper-rectángulo, éste se puede dividir en símplices. La figura 1.4 muestra un ejemplo de como se realiza el particionamiento de un hiper-cubo en seis 3-símplices.

La bisección de un símplex por el lado mayor solo genera un nuevo vértice a evaluar. En el caso de utilizar hiper-rectángulos, en su división por el lado mayor, el número de puntos nuevos a evaluar crece con la dimensión del problema.

La bisección del lado mayor en la división de un símplex regular permite dividir todo el espacio de búsqueda sin solapamiento. En la figura 1.5 se muestra un ejemplo para un 2-símplex.

En [58] Paulavičius y Žilinskas muestran como el uso de símplices para realizar la partición del espacio de búsqueda es conveniente en diferentes problemas de optimización. También se muestran ejemplos de problemas reales en los cuales queda demostrado el mejor comportamiento de particiones basadas en símplices frente a otros tipos

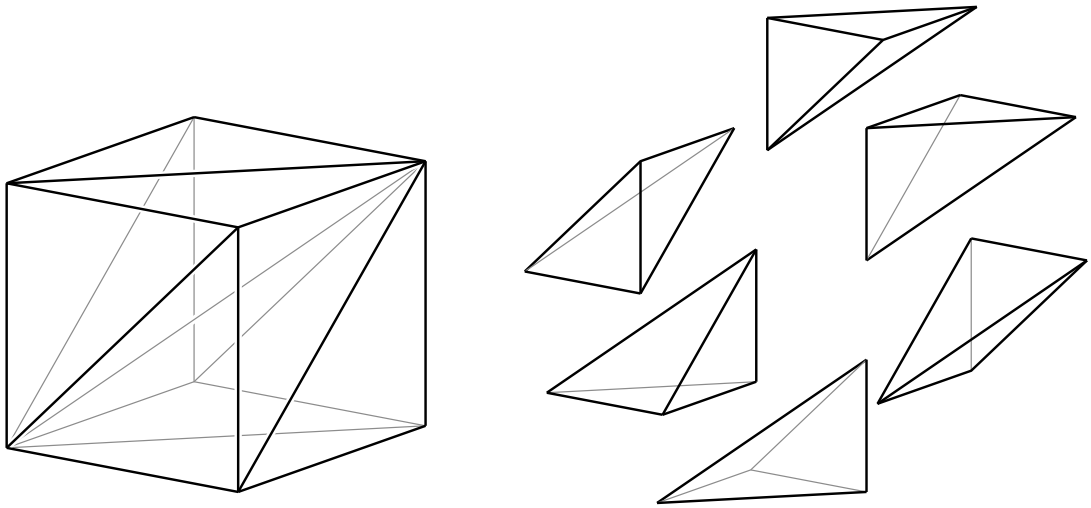


Figura 1.4: División de un hipercubo en símplices.

de particiones. Por ejemplo, se aplica la Optimización Global a regresiones no lineales de mínimos cuadrados, problemas de agrupación estructural de datos (clustering), problemas de emplazamiento de cimientos en construcción o problemas de optimización de Lipschitz. También se ha aplicado para resolver problemas de Blending o Mezclas.

En los problemas de Mezclas, un producto se define como la mezcla de n materiales representados por $x \in \mathbb{R}^n$ [35]. El conjunto de posibles mezclas de los materiales, es decir, el espacio de búsqueda del problema, es el símplex unidad S definido en la ecuación (1.4). En este problema, donde x_j representa la cantidad de material j en la mezcla x, el objetivo del problema es encontrar una mezcla x de menor coste y con menor cantidad de materiales utilizados en ella. En este tipo de problemas es especialmente favorable una división mediante símplices ya que es la propia definición del problema la que establece que el espacio de búsqueda sea un símplex unidad.

Esta tesis se centra en el estudio de la regla de división de algoritmos B&B cuando estos se utilizan para la resolución de problemas de Optimización Global en los que la región de búsqueda está definida por un símplex unidad.

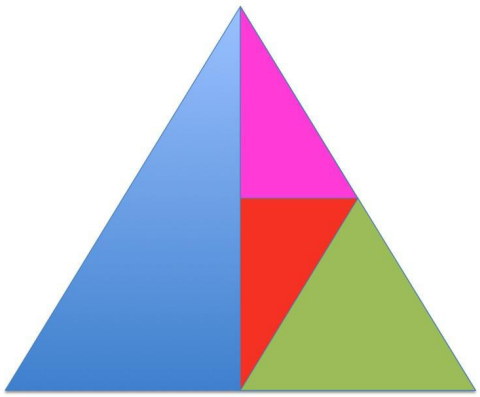


Figura 1.5: División de un 2-símplex por su lado mayor.

1.5 Bisección de un símplex por su lado mayor

A lo largo de esta tesis el conjunto que va a ser refinado es un n-símplex regular al que denotaremos como S_1 y que está escalado de modo que la longitud de sus lados sea 1. Para el cálculo de la longitud de los lados se hará uso de la distancia euclídea. Por lo tanto, S_1 se define como:

Definición 1.

$$S_1 = \left\{ x \in \mathbb{R}^{n+1} \mid \sum_{j=1}^{n+1} x_j = \frac{\sqrt{2}}{2};\ x_j \geq 0 \right\}. \tag{1.5}$$

Un n-símplex es definido por su conjunto de vértices que se denota por $V = \{v_1, \ldots, v_{n+1}\}$, $v_j \in \mathbb{R}^{n+1}$. La figura 1.3 muestra un ejemplo de un 2-símplex regular con lado 1.

La bisección de un símplex por su lado mayor (BLM) es un modo bastante popular de refinamiento dentro del método de elementos finitos, ya que es muy sencillo y puede ser fácilmente aplicable cuando el número de dimensiones es elevado [29]. Este método realiza la división de un símplex mediante el hiperplano que conecta el punto medio del lado mayor del símplex con los vértices opuestos. Un ejemplo se muestra en la figura 1.6.

En el algoritmo 1.2 se muestra el Algoritmo de Refinamiento del Símplex (RS) que bisecciona el símplex inicial iterativamente. En principio, el refinamiento continua indefinidamente, pero aquí se describe el proceso, similar al de B&B, en el que hay un

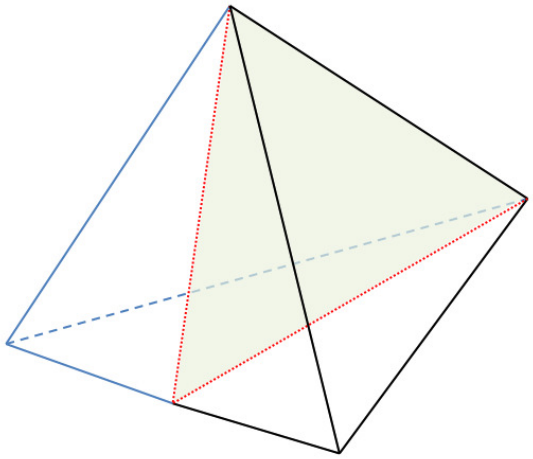


Figura 1.6: Bisección por el lado mayor de un 3-símplex regular.

criterio de parada. El algoritmo continuara con la ramificación hasta que el tamaño de los nodos generados sea menor o igual a una precisión deseada, a la que se denominará ϵ. Se utiliza el índice i para enumerar los símplices del árbol binario resultante. El conjunto Λ proporciona las hojas de árbol que se corresponden con los símplices que aun no han sido refinados. La figura 1.7 muestra el resultado de la aplicación del algoritmo RS a un 2-símplex cuando el criterio de terminación es $\omega(S) \leq \frac{1}{2}$.

Para un 2-símplex, no se requiere la selección entre diferentes lados mayores ya que, o el lado mayor es único, o el símplex a dividir es regular y por tanto la selección de un lado mayor no va a alterar el tamaño del árbol binario generado. Los vértices V se enumeran para tener bien definido el procedimiento SelectLE(). Para la numeración de estos se va a seguir la regla descrita por Mitchell en [49] donde se puede evitar el cálculo de longitudes de lados en un 2-símplex dividiendo siempre por el lado $\{1, 2\}$ y numerando el nuevo vértice como el último del conjunto de vértices del nuevo subsímplex generado. La bisección realizada en el algoritmo 1.3 sigue la misma numeración para el nuevo vértice.

La figura 1.6 muestra el resultado de una primera bisección de un 3-símplex. Independientemente del lado que se seleccione inicialmente los dos subsímplices generados serán simétricos. Como se puede observar estos subsímplices tienen tres lados mayores por lo que se ha de seleccionar uno para realizar su bisección. La figura 1.8 muestra

Algoritmo 1.2 RS(S_1, ϵ)

Requiere: S_1: símplex inicial, ϵ:precisión.

1: $\Lambda := \{1\}$ ▶ Conjunto de índices hoja; los símplices aun no han sido divididos
2: $ns := 1$ ▶ Número de símplices
3: **while** $\Lambda \neq \emptyset$ **do**
4: Extraer un símplex i de Λ
5: **if** $w(S_i) > \epsilon$ **then** ▶ Precisión final no alcanzada
6: $\{j,k\}$:=SelectLE(S_i)
7: $\{S_{2i}, S_{2i+1}\}$:= Bisect(S_i, j, k)
8: Almacenar símplices $2i$ y $2i + 1$ en Λ.
9: $ns := ns + 2$.

Algoritmo 1.3 Bisect(S_i, j, k)

Requiere: S_i: un símplex, j, k: índices de los vértices del lado a dividir.

1: Se usan los vértices v_j y v_k de S para generar $v_{nuevo} := \frac{v_j+v_k}{2}$
2: Los nuevos símplices S_l y S_r heredan las características del símplex origen: $S_l := S_r := S$
3: Eliminar v_j de $V \subset S_l$ ▶ Mover a izquierda vértices $1, \ldots, n$
4: Numerar v_{new} como el vértice $n + 1$ en $V \subset S_l$
5: Eliminar v_k de $V \subset S_r$ ▶ Mover a izquierda vértices $1, \ldots, n$
6: Numerar v_{new} como el vértice $n + 1$ in $V \subset S_r$
7: **return** S_l, S_r

como la selección de un lado mayor u otro da lugar subsímplices diferentes.

El número de símplices del árbol binario finito que se genera con el algoritmo 1.2 depende de cómo de rápido decrece el tamaño de los símplices conforme se desciende en el árbol.

1.6 Preguntas que surgen en esta investigación

Durante el proceso de bisección iterativa de un símplex regular en algoritmos de B&B se genera un árbol binario. Estamos interesados en conocer el tamaño de un subárbol a partir de un nodo cualquiera del árbol para así poder realizar una estimación de la carga computacional pendiente. El tamaño del árbol dependerá de cómo disminuye el tamaño

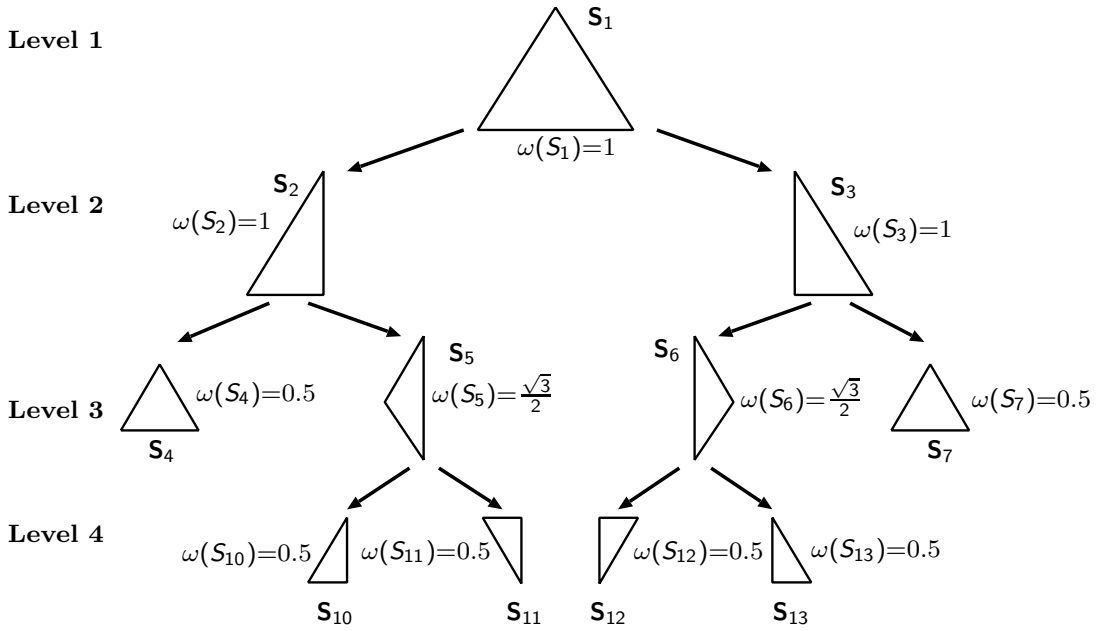


Figura 1.7: Árbol binario generado por el algoritmo RS en un 2-símplex con $\epsilon = \frac{1}{2}$.

de los subproblemas conforme se aplica la regla de división. Kearfott presenta un límite superior del tamaño de los subproblemas cuando se usa el método de bisección del lado mayor (BLM) en el siguiente teorema [43]:

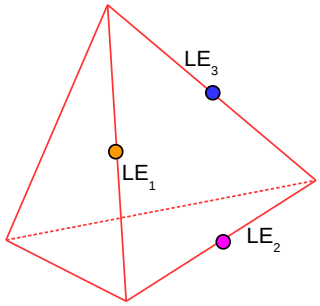
Teorema 1.1. *Sea S_0 un n-símplex, p un entero positivo cualquiera y S_p cualquier n-símplex que se ha producido tras p BLM de S_0. Entonces, el diámetro (o lado mayor) de S_p nunca será mayor que $(\frac{\sqrt{3}}{2})^{\lfloor \frac{p}{n} \rfloor}$ veces el diámetro de S_0, donde $\lfloor \frac{p}{n} \rfloor$ es el mayor entero menor o igual a $\frac{p}{n}$.*

Kearfott también mostró que, para un símplex S, los diámetros se reducían un factor 2 cada n iteraciones para $p > n$. Stynes mejoró la cota de Kearfott para 2-símplices [72]. La cota que se propuso es

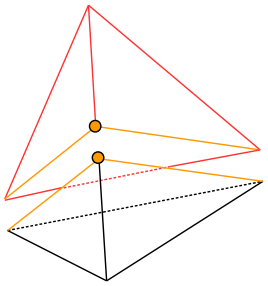
$$S_{2p} \leq \frac{\sqrt{3}}{2} \left(\frac{1}{2}\right)^{\frac{p}{2}-1} \tag{1.6}$$

Por ejemplo, para p=6, la cota de Kearfott es

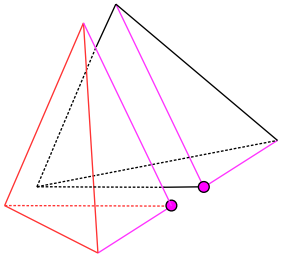
$$S_6 \leq \frac{\left(\sqrt{3}\right)^3}{(2)^3} \tag{1.7}$$



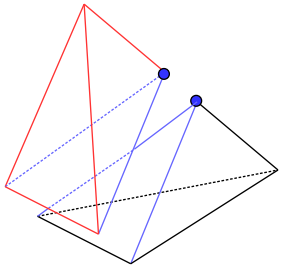
(a) Lados mayores en un 3-símplex en el nivel 2 del árbol



(b) División por LE_1



(c) División por LE_2



(d) División por LE_3

Figura 1.8: 3-Símplices generados dependiendo del lado mayor seleccionado.

mientras que la cota de Stynes es

$$S_6 \leq \frac{\left(\sqrt{3}\right)}{(2)^3} \tag{1.8}$$

lo cual es $\left(\sqrt{3}\right)^2$ veces menor que la cota encontrada por Kearfott. Nuestro propósito es hallar los valores exactos del tamaño del árbol binario y no cotas superiores de su tamaño. El problema que surge es que para símplices de dimensión mayor que tres, la selección del lado mayor no es única, es decir, aparecen varios lados mayores que pueden ser divididos (véase la figura 1.8). Tradicionalmente, por comodidad se ha seleccionado el primer lado mayor según la ordenación de los vértices del símplex.

Pregunta 1.1. *¿Existe una heurística de selección del LM a dividir que genere un árbol menor que la selección del primer LM?*

Para responder esta pregunta, en el capítulo 2 se van a plantear diferentes heurísticas de selección del lado mayor.

Pregunta 1.2. *¿Alguna de esas heurísticas obtiene el árbol de menor tamaño?*

Para poder responder a la pregunta 1.2, en el capítulo 3 se han desarrollado algoritmos secuenciales para responder a la siguiente pregunta:

Pregunta 1.3. *¿Cuál es el tamaño del menor árbol binario generado mediante BLM de un símplex regular?*

El problema de encontrar el menor tamaño del árbol es un problema de Optimización Combinatoria cuyo coste computacional crece exponencialmente con la dimensión y la precision requeridas. Debido a la mayor facilidad de desarrollo de programas paralelos en memoria compartida, se plantea la siguiente pregunta:

Pregunta 1.4. *¿Cuál es el modelo de programación paralelo más eficiente para resolver este problema en sistemas multicore de memoria compartida?*

En el capítulo 4 se estudian algortimos paralelos basados en TBB y Pthreads con un número estático y dinámico de hebras.
Para hacer una estimación del tamaño de un subárbol a partir de un nodo, el tener un árbol con el menor número de formas de símplices reduciría el número de posibles subárboles y por lo tanto, facilitaría el cálculo de una estimación de su tamaño.

Pregunta 1.5. *¿Cuál es el menor número de formas de los subsímplices en el refinamiento de un símplex regular mediante BLM?*

El capítulo 5 está dedicado a resolver la pregunta 1.5.
En el refinamiento de un símplex mediante BLM, hay que realizar repetidamente el cálculo de distancias para conocer los lados mayores susceptibles de ser divididos. Para reducir el coste computacional de este tipo de algoritmos se puede proporcionar una secuencia de lados mayores a dividir que genere el menor árbol binario. Estamos interesados en encontrar la secuencia recurrente de menor tamaño.

Pregunta 1.6. *¿Cuál es la secuencia recurrente de lados mayores a dividir y de menor tamaño para obtener el árbol binario de menor tamaño?*

En el capítulo 6 se estudian varios algoritmos que conjuntamente dan una respuesta a la pregunta 1.6.

1.7 Conclusiones

Este capítulo ha comenzado describiendo los problemas de Optimización Global y se han presentado los algoritmos de Ramificación y Acotación como método para resolver de este tipo de problemas. Se ha definido el símplex como espacio de búsqueda en los algoritmos de Ramificación y Acotación y tras describir las características principales de los símplices se ha descrito el método BLM como método de refinamiento. Se ha mostrado como el refinamiento de un símplex regular mediante BLM genera subsímplices con varios lados mayores para dimensiones mayores que tres y que la selección de uno u otro a la hora de realizar BLM afecta en la forma de los símplices y el tamaño del árbol binario generado. Este capítulo concluye presentando las preguntas que han surgido durante el desarrollo de esta tesis.

Capítulo

2

Heurísticas de selección del lado mayor a dividir para reducir el tamaño del árbol binario generado

En este capítulo se va a dar respuesta a la pregunta 1.1: ¿existe una heurística de selección del LM a dividir que genere un árbol menor que la del primer LM? En la sección 2.1 se proponen diferentes heurísticas que tienen como finalidad reducir el número total de símplices del árbol. En la sección 2.2 se presentan los resultados para diferentes dimensiones del problema.

2.1 Descripción de las heurísticas

Para símplices de dimensión mayor que tres, la selección del lado mayor no es única, es decir, aparecen varios lados mayores que pueden ser divididos. Tradicionalmente, por comodidad se ha seleccionado como lado mayor a dividir el primero según la numeración de los vértices usada en el algoritmo 1.3. En este apartado se van a plantear diferentes heurísticas de selección del lado mayor para intentar reducir el número total de símplices del árbol binario resultante [24] [25].

2.1.1 Primer Lado Mayor, LEB_1

Esta es la heurística más sencilla y se usará como base con la que se van a comparar el resto. Se denomina LEB en referencia a las siglas en inglés de bisección por el lado

mayor, Longest Edge Bisection. Se basa en seleccionar el primer lado mayor que viene determinado por los índices de los vértices que lo componen, según el algoritmo 1.3.

2.1.2 Lado mayor con menores ángulos en sus vértices, LEB$_\alpha$

El método LEB$_\alpha$ para la selección del lado mayor a dividir trata de evitar la formación de símplices con forma "mala" o "pobre" [23]. Una de las características de este tipo de símplices es que forman ángulos de valor muy reducido entre los lados de algunos de sus vértices. Por lo tanto, este método analizará los ángulos que forman los lados de un símplex en sus vértices y seleccionará como lado mayor a dividir aquel que provoque los ángulos que se van a dividir durante el proceso sean lo más grandes posible.

El método LEB$_\alpha$ hace uso del cálculo de ángulos que se ha expuesto en (5.7).

El comportamiento de LEB$_\alpha$ como método de selección del lado mayor dentro de un algoritmo de B&B ha sido probado en [34].

2.1.3 Lado mayor con su punto medio lo más alejado posible del centroide del símplex, LEB$_C$

Esta regla selecciona como lado mayor a dividir aquel cuyo su punto medio esté lo más alejado del centroide del símplex. Se pretende que los símplices hijos tengan los centros de los lados mayores más cerca del centroide que en el símplex padre, promoviendo de esta manera subsímplices con forma más redondeada.

La distancia del punto medio del lado $\{v_i, v_j\}$ al centroide C viene dado por

$$\|\frac{v_i + v_j}{2} - C\|. \tag{2.1}$$

Por lo tanto, la heurística LEB$_C$ selecciona como lado mayor a dividir $\{v_i, v_j\}$ que cumple la condición:

$$\max_{i,j} \|\frac{v_i + v_j}{2} - C\|. \tag{2.2}$$

2.1.4 Lado mayor con vértices que hayan intervenido menos en procesos de bisección, LEB$_W$

En esta heurística se trabaja utilizando pesos w_i. Cada vértice va a tener un peso asociado al número de veces que ha participado en un proceso de bisección. De este modo

se va a calcular, para cada lado mayor, la suma del peso de sus vértices y se dividirá el lado mayor cuya suma de pesos sea menor, es decir, se pretende dividir el lado mayor que menos haya intervenido en procesos de bisección. A cada vértice v_i se le asigna un peso $w_i \in \{0, \ldots, n-1\}$ y durante el proceso de refinamiento el peso de un vértice v_i será incrementado si ese vértice v_i forma parte del lado mayor que se ha dividido. Inicialmente los vértices tendrán un peso 0. Los nuevos vértices que van apareciendo como parte del proceso comenzarán teniendo un peso 1 ya que han participado en un proceso de división.

El peso de un vértice v_i es incrementado en 1 si se divide el lado $\{v_i, v_j\}$. En ese caso v_i obtiene un peso $w_i := (w_i + 1) \mod n$. LEB$_W$ seleccionará como lado mayor a dividir aquel cuya suma de pesos en sus vértices sea la menor.

2.1.5 Combinación de LEB$_W$ con un segundo criterio , LEB$_{W_2}$

Aun a pesar de utilizar LEB$_W$ existen casos en los que varios lados mayores poseen una suma de pesos igual. Para tratar de elegir uno de entre esos lados mayores en esta heurística se han añadido dos cambios significativos. El primero de ellos es el cálculo del peso w_i que ahora se realiza añadiendo 1 al valor del peso del vértice al que se sustituye. El segundo cambio es la adición de un nuevo criterio de selección que seleccionará de entre todos los lados mayores con igual suma de pesos, aquel que sea más antiguo, es decir, que haya sido creado en un proceso de bisección más antiguo. Para establecer la antigüedad del proceso de bisección se utiliza un contador que sencillamente cuenta las divisiones realizadas durante el proceso. Cada vez que un nodo se divide, aumenta el valor del contador. Cada nodo generado almacena el valor del contador en el momento de su creación. De este modo se podrá establecer el orden de creación de todos los nodos.

2.1.6 Lado mayor con el punto medio más alejado del resto de los vértices, LEB$_M$

Esta heurística selecciona como lado mayor a dividir aquel cuyo punto medio esté más alejado del resto de los vértices:

$$\max_{i,j} \max_{k} ||\frac{v_i + v_j}{2} - v_k|| \tag{2.3}$$

Ya que los lados que parten de un vértice nuevo para la pareja de subsímplices creados en una bisección son iguales, esta heurística consigue priorizar en la división la creación de símplices más compactos.

2.2 Resultados

Para todas las heurísticas presentadas puede darse el caso de que existan varios lados mayores cumpliendo con los requisitos exigidos. En tal caso, el lado mayor seleccionado para ser dividido será el primero de ellos según la ordenación de vértices del algoritmo 3.

La tabla 2.1 muestra el número de símplices generados utilizando cada heurística para diferentes valores de n y de ϵ. Para esos valores también se proporciona el número de símplices del árbol de menor tamaño, obtenido utilizando el algoritmo 4.8 de la sección 4.3.2. Se han elegido valores de precisión $\epsilon = \frac{1}{2^k}$ suficientemente pequeños para obtener un árbol de búsqueda de un tamaño grande, pero razonable.

Tabla 2.1: Número de símplices del árbol binario obtenido por refinamiento de un símplex regular de tamaño unidad utilizando diferentes heurísticas.

	$n = 3$ y $\epsilon = 0,015625$	$n = 4$ y $\epsilon = 0,0625$	$n = 5$ y $\epsilon = 0,25$
LEB_1	2.114.687	1.577.047	79.035
LEB_α	1.398.271	1.196.327	60.191
LEB_C	1.398.271	1.189.213	57.535
LEB_W	1.822.377	1.291.883	50.595
LEB_{W_2}	1.766.521	1.252.505	49.107
LEB_M	1.398.271	1.185.751	55.791
Valor mínimo	1.398.271	877.959	36.319

Los resultados muestran que aunque las reglas de selección de lado mayor LEB_α (2.1.2), LEB_C (2.1.3), LEB_W (2.1.4), LEB_{W_2} (2.1.5) y LEB_M (2.1.6) son computacionalmente más complejas que LEB_1 (2.1.1), el tiempo de ejecución global de algoritmo

de B&B cuando se utiliza cualquiera de ellas se reducirá significativamente ya que se decrementa en gran número el total de símplices evaluados.

En la tabla 2.1, LEB_{α}, LEB_{C} y LEB_{M} proporcionan el mejor resultado para n=3. Entre ellos, la nueva propuesta LEB_{M} es la que presenta un menor coste computacional. LEB_{M} es además la heurística que menor número de símplices obtiene para n=4. Sin embargo, ese no es el caso para n=5, donde es LEB_{W_2} es la heurística con un mejor resultado. La ambigüedad de las heurísticas crece conforme lo hace n [24], es decir, a medida que aumentamos n aparecen más lados mayores que satisfacen los requerimientos de la heurística y por tanto los criterios de selección usados en cada heurística son mejorables.

2.3 Conclusiones

En este capítulo, se han desarrollado varias heurísticas de selección de lado mayor que tratan de reducir el número de símplices que tiene el árbol binario que aparece durante este proceso. La respuesta a la pregunta 1.1 es que todas las heurísticas presentadas mejoran el tamaño del árbol generado por la heurística LEB_1. La conclusión del estudio realizado en este capítulo es que la mejor heurística varia dependiendo de la dimensión del problema.

Respecto a la pregunta 1.2: ¿alguna de esas heurísticas obtiene el árbol de menor tamaño?, la tabla 2.1 muestra que utilizando cualquiera de las heurísticas presentadas, sigue habiendo ambigüedad en la selección del lado mayor y que el tamaño del árbol obtenido no es el menor para $n > 3$. Por lo tanto, el encontrar una heurística que consiga el menor árbol es una cuestión todavía sin resolver.

Capítulo

3

Algoritmos secuenciales para encontrar el tamaño del menor árbol binario en el refinamiento de un símplex

Durante el proceso de refinamiento de un símplex regular utilizando BLM aparece un árbol binario. Dependiendo de las combinaciones de lados mayores seleccionados para ser divididos, el tamaño final del árbol binario varía.

En este capítulo se da respuesta a la pregunta 1.3: ¿cuál es el tamaño del menor árbol binario generado mediante BLM de un símplex regular? Conocer el tamaño del menor árbol permitirá verificar que las nuevas propuestas de selección de lado mayor a dividir conducen realmente a la obtención del menor árbol.

En este capítulo se describen algoritmos para determinar el tamaño del menor árbol binario generado usando BLM como regla de división de los símplices. El algoritmo debe comprobar las posibles opciones de división de lado mayor, una por cada lado mayor, teniendo en cuenta solo aquellas opciones que generan un árbol binario de menor tamaño. Este proceso comenzaría su ejecución en el n-símplex regular inicial S_1. Para cada uno de los dos subsímplices generados se calcula el tamaño del subárbol generado por la posible división de cada lado mayor, devolviendo el tamaño del menor de ellos. Los algoritmos tendrán el símplex inicial y la precisión requerida como parámetros de entrada y devuelven el número de símplices del árbol mínimo.

Para conocer el tamaño del menor árbol binario, se necesita un algoritmo de búsqueda exhaustivo que pruebe todas las posibilidades de división por lado mayor de cada

subsímplex. La búsqueda debe evitar simetrías, es decir, en caso de existir lados mayores que den lugar a subárboles de igual tamaño, tan solo se ha de comprobar uno de ellos. Esto ocurre por ejemplo, si los dos símplices resultantes de la división de un símplex padre son simétricos, por lo que tan solo será necesario obtener el número de símplices del subárbol de uno de ellos y el número de símplices bajo el árbol padre será dos veces el valor calculado.

El algoritmo de búsqueda del tamaño del menor árbol binario en el refinamiento de un símplex comprueba todas las divisiones que se pueden hacer en todos los símplices que aparecen en el proceso. Para resolver este problema se crea un árbol en el denominado modo *hacia delante* y se poda en el modo denominado *hacia atrás*. El modo *hacia delante* solo se utiliza para visitar las ramas del árbol. En el modo *hacia atrás*, el árbol se recorre desde las hojas a la raíz determinando las ramas de menor tamaño. Dicho proceso se puede describir como un algoritmo de B&B con las siguientes reglas:

División: en modo *hacia delante*, se debe comprobar cada lado mayor del símplex. Para cada lado mayor del símplex a tratar, la regla de división generará un par de nuevos subproblemas en el proceso de bisección. Por lo tanto, el árbol de búsqueda, al que se va a denominar árbol general, puede tener nodos con diferentes factores de ramificación. Por ejemplo, un símplex con tres lados mayores va a dar lugar a seis nuevos subproblemas.

Selección: en modo *hacia delante*, se va a seleccionar como siguiente subproblema a tratar aquel cuyo símplex se encuentre en un nivel más profundo del árbol general. Por lo tanto, se realizará una búsqueda en profundidad (ver sección 1.2). Se hace de este modo porque este tipo de búsquedas tienen menos requerimientos de memoria que otros tipos de selección [38].

Acotación: en modo *hacia atrás*, se pueden calcular cotas superior e inferior del tamaño de los subárboles binarios de un nodo cuando todos los subárboles generados por diferentes BLM han sido creados.

Rechazo: en modo *hacia atrás*, las divisiones de lados mayores en símplices que lleven a la obtención de subárboles mayores no serán considerados para el cálculo de árbol binario de menor tamaño.

Terminación: en modo *hacia delante*, un símplex con un tamaño menor a una precisión ϵ dada, no volverá a ser dividido. En modo *hacia atrás*, el algoritmo fina-

lizará cuando se calcule la regla de acotación para el nodo raíz. La solución del problema será entonces la cota inferior calculada para el nodo raíz más uno.

La consecuencia del uso de la búsqueda en profundidad en el árbol general es que cada rama, y por tanto el árbol general que contiene todos los posibles árboles binarios, debe ser visitado completamente para poder determinar el tamaño del árbol de menor tamaño.

Otro enfoque se puede basar en visitar cada uno de los árboles binarios y descartar la visita del árbol binario cuando su tamaño sea mayor que el tamaño más pequeño encontrado hasta el momento. Este enfoque no es atractivo porque el número árboles binarios crece rápidamente con la dimensión n y con el aumento de la precisión. En este capítulo se van a presentar dos enfoques diferentes para el algoritmo B&B descrito anteriormente. Un enfoque es recursivo y el otro iterativo y van a ser la base de las versiones paralelas. En la versión recursiva, la función principal divide un símplex y se llama a si misma para procesar los nuevos símplices hasta que el ancho de un símplex es menor que la precisión deseada. En el enfoque iterativo, se utiliza memoria dinámica para establecer las relaciones entre los nodos del árbol general, en vez de usar las llamadas a procedimiento, como hace la versión recursiva.

3.1 Algoritmo R-STS (Recursive - Smallest Tree Size)

El algoritmo 3.1 es un procedimiento secuencial recursivo para calcular el número de símplices de uno de los árboles binarios de menor tamaño creados mediante el método de bisección del lado mayor. El algoritmo 3.1 comienza comprobando si la anchura del símplex actual S_i es menor que ϵ. En tal caso, devuelve 1 como la longitud de su árbol (línea 2). En otro caso, se divide S_i por cada uno de sus lados mayores LE_h (línea 3), generando cada pareja de símplices hermanos S_{2i} y S_{2i+1} de un árbol binario. El tamaño del árbol de cada hermano se determina por una llamada recursiva (líneas 5 y 6). La suma de los tamaños de los subárboles de los símplices hermanos se almacena en r_h. Finalmente, se devuelve como resultado el menor valor de $r_h + 1$ de los lados divididos h (línea 7), porque S_i se debe contar en el resultado.

Algoritmo 3.1 R-STS (S_i, ϵ)

Requiere: S_i: símplex, ϵ: precisión

1: **if** $w(S_i) \leq \epsilon$ **then**
2: **return** 1
3: **for** cada lado mayor LE_h de S_i **do**
4: $\{S_{2i}, S_{2i+1}\}$:=Bisect(S_i,LE_h)
5: $r_{2i,h}$:= R-STS (S_{2i}, ϵ)
6: $r_{2i+1,h}$:= R-STS (S_{2i+1}, ϵ)
7: $r_h = r_{2i,h} + r_{2i+1,h}$
8: **return** $1 + \min_h r_h$

3.2 Algoritmo I-STS (Iterative - Smallest Tree Size)

Este algoritmo utiliza memoria dinámica para almacenar los enlaces entre los nodos del árbol general en lugar de utilizar para ello las llamadas recursivas al procedimiento. Al igual que en el algoritmo 3.1, se comprueba cada división por el lado mayor para obtener el mínimo tamaño del árbol binario resultante.

Un nodo N del árbol general tiene los siguientes componentes:

- S: El n-símplex que va a ser analizado. Almacena los vértices y el índice del símplex en el árbol binario.
- CC: Variable booleana indicando si los nodos hijos han sido creados.
- nLE: El número de lados mayores (LEs) de S.
- nCN: Número de nodos hijo, con $nCN = 2 * nLE$. Se generan dos símplices hijo por cada lado mayor LM de S.
- CN: Vector que contiene punteros a los nodos hijo. Son punteros a cada uno de los nodos que se generan por bisección del símplex S para todos y cada uno de sus lados mayores. $CN = \{CN_i\},\ i = 0, \ldots,\ nCN - 1$.
- $CSTS$: Vector con el tamaño de cada subárbol correspondiente a cada uno de los nodos hijos.
- STS: Tamaño del subárbol. Después de que se hayan calculado todos los valores del vector $CSTS$, se calcula STS como uno más la suma de la pareja de hijos con menor suma de tamaño de subárboles.
- PN: Puntero al nodo padre del nodo actual N.

- $IinPN$: Posición del nodo N en el vector $CSTS$ del nodo padre. Se utiliza para almacenar el valor de STS en la posición correspondiente del vector $CSTS$ del nodo padre al que se accede haciendo uso del puntero PN.

La Figura 3.1 muestra la estructura de datos de un nodo N para el caso de un símplex S que tiene tres lados mayores. Los punteros que se observan con líneas continuas representan la conexión con el nodo padre mientras que los que aparecen con líneas discontinuas representan la conexión con los nodos hijos.

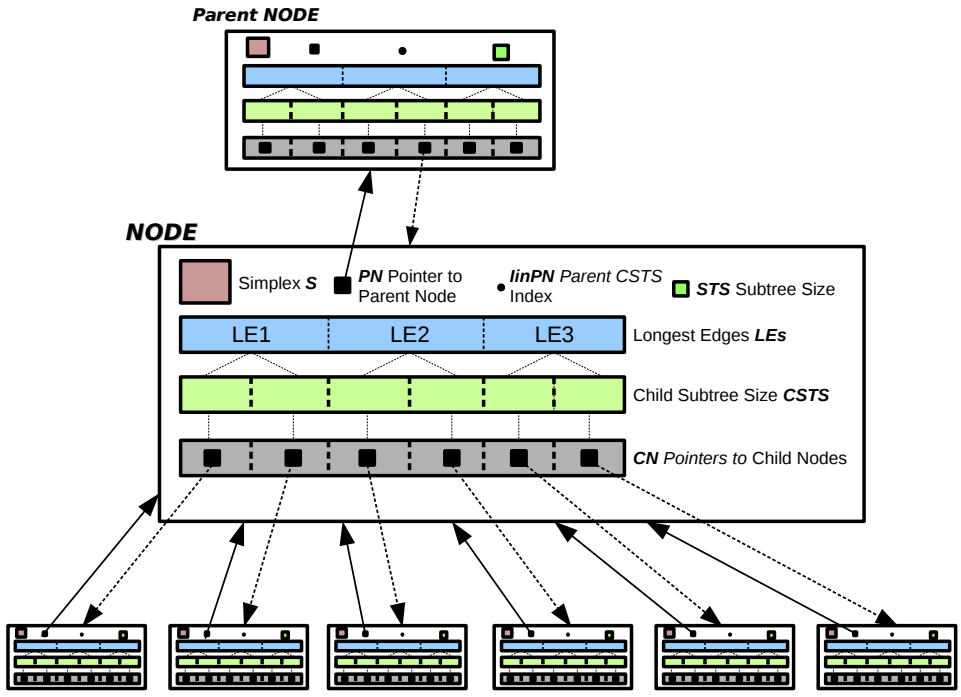


Figura 3.1: Estructura de datos de tipo Nodo

El algoritmo 3.2 crea un nodo con el nuevo símplex, un enlace a su nodo padre y su posición en el vector $CSTS$ del nodo padre ($IinPN$). El tamaño del subárbol que tiene como raíz este nodo (STS) se inicializa a 0 porque aún se desconoce su valor. CC se inicializa a falso porque los nodos hijos aun no han sido creados.

El algoritmo 3.3 crea los nodos hijo dividiendo el símplex S_i por cada uno de sus lados mayores LMs (línea 4). Se crea un nuevo nodo para cada hijo siempre y cuando

Algoritmo 3.2 CreaNodo (S_i, PN, p)

Requiere: S_i: símplex, PN: nodo padre, p: posición en $CSTS$ del nodo padre

1: Crea nuevo nodo N con S_i y PN.
2: $IinPN := p$
3: STS:=0 ▶ El tamaño aun es desconocido
4: CC:=**false** ▶ Sus nodos hijo aun no han sido creados
5: **return** N

el tamaño del hijo sea mayor que ϵ (líneas 8 y 13), inicializando el tamaño de subárbol a desconocido (líneas 9 y 14). En caso contrario, el tamaño de su subárbol es conocido y el correspondiente elemento de $CSTS$ se establece a 1 (líneas 6 y 11).

Algoritmo 3.3 CreaNodosHijo (N)

Requiere: N: un nodo creado.

1: nLE:= NúmerodeLE($S_i \in N$) ▶ Número de lados mayores
2: $nCN := 2 \cdot nLE$ ▶ Número de nodos hijo
3: **for** $h \leftarrow 0, nLE$ **do**
4: $\{S_{2i}, S_{2i+1}\}$:= Biseccciona(S_i, LE_h) ▶ Dos hijos se generan por cada LE_h
5: **if** $\omega(S_{2i}) \leq \epsilon$ **then**
6: $CSTS_{2h} := 1$ ▶ No se crea nodo hijo
7: **else**
8: CN_{2h} := CreaNodo(S_{2i}, N, $2h$) ▶ Crea un nodo para el hijo de la izquierda
9: $CSTS_{2h} := 0$ ▶ Se desconoce el tamaño del subárbol
10: **if** $\omega(S_{2i+1}) \leq \epsilon$ **then**
11: $CSTS_{2h+1} := 1$ ▶ No se crea nodo hijo
12: **else**
13: CN_{2h+1} := CreaNodo(S_{2i+1}, N, $2h+1$) ▶ Crea un nodo para el hijo de la derecha
14: $CSTS_{2h+1} := 0$ ▶ Se desconoce el tamaño del subárbol

El algoritmo secuencial iterativo para el cálculo del árbol binario con el número mínimo de símplices (I-STS) se presenta en algoritmo 3.4. Tras comprobar que el valor de ϵ es menor que el tamaño del símplex inicial (línea 1), se crea el primer nodo con el n-símplex inicial S_1 (línea 3). En la primera llamada a CreaNodo() el puntero al nodo padre se establece a NULL y p a 0 ya que el nodo raíz no tiene nodo padre. El bucle del algoritmo (línea 4) hace un recorrido del árbol utilizando una búsqueda en profundidad. Para un nodo con un símplex cuyo tamaño sea mayor que ϵ, se crean los nodos hijos,

si aun no han sido creados (línea 6). Del conjunto de los nodos hijo, el siguiente a ser procesado será aquel que no tenga ningún hijo creado (líneas 7 y 9). Una vez que todos los nodos hijos han sido visitados (línea 11), se calcula el tamaño del subárbol menor con raíz en este nodo (línea 12). El algoritmo actualiza el elemento correspondiente de $CSTS$ del nodo padre (línea 14) y se desplaza al nodo padre (línea 15), siempre y cuando el nodo actual no sea el nodo inicial. Si el nodo actual es el nodo inicial, el algoritmo devuelve el valor del número de símplices del árbol binario de menor tamaño y finaliza (línea 18).

Algoritmo 3.4 I-STS (S_1, ϵ)

Requiere: S_1: símplex inicial, ϵ: precisión

1: **if** $\omega(S_1) \leq \epsilon$ **then**
2: **return** 1
3: N := CreaNodo(S_1, NULL, 0) ▶ Ver algoritmo 3.2
4: **while true do**
5: **if** CC = **false then** ▶ Nodos hijo no han sido creados
6: CreaNodosHijo(N) ▶ Ver algoritmo 3.3
7: $j := \exists$ a $CSTS_j = 0, -1$ en otro caso ▶ Existe un hijo pendiente de procesar
8: **if** $j \geq 0$ **then**
9: $N := CN_j$ ▶ Desplazamiento al nodo hijo
10: **continua**
11: **else** ▶ Todos los tamaños de los subárboles han sido calculados
12: $STS := \min_k\{CSTS_k + CSTS_{k+1}\} + 1,\ k \mod 2 = 1$
13: **if** $S_i \neq S_1$ **then**
14: $CSTS_{IinPN} \in PN := STS$ ▶ Actualizar $CSTS$ del nodo padre
15: $N := PN$ ▶ Desplazamiento al nodo padre y liberación del antiguo nodo N
16: **continua**
17: **else**
18: **return** STS ▶ El algoritmo finaliza

3.3 Estrategias para reducir la búsqueda

Los requerimientos de computación y de memoria de este tipo de algoritmos es muy alto para instancias grandes del árbol general. Algunas estrategias para reducir la búsqueda han sido tomadas en cuenta.

Mejora 3.1.

En la bisección de un símplex regular S el tamaño de los subárboles es independiente del lado seleccionado. Por lo tanto, solo es procesado el primer lado mayor. Por simplicidad, esta comprobación no se ha incluido en la selección del lado mayor a dividir (algoritmo 3.1 línea 3, y algoritmo 3.3 línea 3).

Mejora 3.2. *Dos símplices hijo pueden ser gemelos, es decir, pueden ser simétricos y por tanto tendrían la misma forma y tamaño [26]. Los subárboles correspondientes a símplices gemelos van a tener el mismo tamaño. Por lo tanto, tan solo uno de ellos necesita ser procesado. Esta mejora se implementa después de generar el par de símplices hijo en el algoritmo 3.1 línea 4 y en el algoritmo 3.3 línea 4, pero por simplicidad, no se ha incluido en la descripción de los algoritmos.*

Se puede ver un ejemplo de la mejora 3.2 para un 3-símplex en el paso 1 de la figura 4.1. Los círculos representan los símplices y los cuadrados representan las opciones para sus lados mayores. Siguiendo la mejora 3.1 el primer símplex es dividido tan solo una vez porque es regular. Los símplices que han sido tachados no son procesados porque tienen un hermano gemelo.

Otro modo de reducir la búsqueda es comprobar la existencia de pares de parejas de hermanos con la misma forma y tamaño. Cada par de hermanos va a producir árboles del mismo tamaño. Saber que dos parejas de hermanos tienen la misma forma no es una tarea fácil [26] y será abordada en estudios futuros.

3.4 Resultados

Los algoritmos se han ejecutado en las siguientes máquinas:

- BullX (16 cores): formado por 2 procesadores Intel® Xeon® E5-2620 (Sandy Bridge) de 8 núcleos. La frecuencia de trabajo del procesador es de 2 GHz. Cada núcleo tiene 32KB de memoria caché L1 y 256KB de memoria caché L2. Cada procesador dispone de 20 MB de memoria cache L3 compartida entre todos sus núcleos y la máquina tiene 64 GB de memoria RAM. La versión del compilador gcc utilizada en esta máquina es la 4.8.1 con la especificación C++11 para C++.

- Aloe (32 cores): formado por 2 procesadores Intel® Xeon® E5-2698 V3 (Haswell) de 16 núcleos. La frecuencia de trabajo del procesador es de 2.3 GHz. Cada núcleo tiene 32KB de memoria caché L1 y 256KB de memoria caché L2. Cada procesador dispone de 40 MB de memoria cache L3 compartida entre todos sus núcleos y la máquina tiene 256 GB de memoria RAM. La versión del compilador gcc utilizada en esta máquina es la 4.8.2 con la especificación C++11 para C++.

El problema que se quiere resolver presenta un bajo coste computacional por símplex, pero un alto coste en términos de gestión dinámica de la memoria, que es necesaria para procesar el árbol binario cuando el número de símplices que contiene es alto. Por lo tanto, los efectos del uso de un gestor de memoria dinámica u otro en los tiempos de ejecución del algoritmo van a ser muy altos. GCC tiene su propio gestor de memoria dinámica, pero también se han probado otros:

- Thread-Caching Memory Allocator (TCMAlloc) de Google Performance Tools: es una librería desarrollada por Google optimizada para situaciones de alta concurrencia. El método a través del cual este gestor realiza las asignaciones de memoria sin bloqueos es el uso de una cache para las hebras.
- Intel Threading Builing Blocks Memory Allocator (TBBMAlloc) de Intel: es una librería de gestión de memoria diseñada por Intel específicamente para ser utilizada con su entorno de programación TBB. Sin embargo, puede ser utilizada en cualquier programa que haga uso de memoria dinámica. Se centra en la optimización de dos aspectos muy importantes de la computación paralela, conseguir escalabilidad y evitar false sharing.
- Lockless Memory Allocator (LLalloc) de *Lockless Inc.*: es un gestor de memoria dinámica de altas prestaciones diseñado por la empresa Lockless Inc. A diferencia de otros gestores de memoria, libera memoria al sistema operativo cuando esta ya no se necesita más.

Las versiones de las librerías de los diferentes gestores de memoria utilizados son: TCMAlloc versión 2.1, TBBMAlloc 4.2 y LLalloc versión 1.4.

Los resultados se han obtenido para las siguientes instancias del problema:

- I3: 3-símplex con $\epsilon = 0,025$.
- I4: 4-símplex con $\epsilon = 0,125$.
- I5: 5-símplex con $\epsilon = 0,25$.
- I6: 6-símplex con $\epsilon = 0,5$.

La tabla 3.1 muestra una comparación del tiempo de ejecución requerido por el algoritmo secuencial recursivo Alg. 3.1 para resolver las instancias I3 e I4 en la máquina BullX utilizando diferentes gestores de memoria. Aunque el algoritmo 3.1 es secuencial, realiza asignaciones y liberación de memoria para los símplices generados y eliminados cuando se recorre el árbol general.

Tabla 3.1: Tiempo de ejecución del algoritmo 3.1 en segundos para las instancias I3 e I4 utilizando diferentes librerías de gestión de memoria dinámica en la máquina BullX.

Instancia	GCC	TCMalloc	TBBMalloc	LLAlloc
I3	40,85	45,42	41,32	30,36
I4	192,61	214,75	193,49	152,71

Dependiendo del problema a resolver unos gestores de memoria pueden comportarse mejor que otros. Sin embargo, para nuestro caso de estudio, Lockless Allocator ha resultado ser el gestor que más reduce el tiempo de ejecución de los algoritmos. Por lo tanto, de ahora en adelante se va a utilizar solamente Lockless Allocator.

La tabla 3.2 muestra una comparación del tiempo de ejecución requerido por el algoritmo secuencial recursivo Alg. 3.1 y por el algoritmo secuencial iterativo Alg. 3.4 para resolver las instancias I3, I4, I5 e I6 en la máquina BullX.
La tabla 3.3 muestra resultados similares a los que aporta la tabla 3.2, pero para la máquina Aloe.

En las tablas 3.2 y 3.3 se observa que la versión R-STS obtiene unos tiempos de ejecución menores. Esto es debido a que I-STS debe gestionar una estructura de datos de tamaño mucho mayor a la que utiliza R-STS. En algoritmos con un alto coste en términos de gestión dinámica de la memoria el tamaño de las estructuras de datos a gestionar afecta mucho al tiempo de ejecución del algoritmo.

Tabla 3.2: Tiempo de ejecución de los algoritmos 3.1 y 3.4 en segundos para las instancias I3, I4, I5 e I6 en la máquina BullX.

	I3	I4	I5	I6
R-STS	30,36	152,71	12.301,92	15.566,24
I-STS	39,55	186,45	14.862,73	21.328,00

Tabla 3.3: Tiempo de ejecución de los algoritmos 3.1 y 3.4 en segundos para las instancias I3, I4, I5 e I6 en la máquina Aloe.

	I3	I4	I5	I6
R-STS	11,07	57,75	4.653,16	7.056,76
I-STS	16,31	77,44	5.954,82	8.918,79

3.5 Conclusiones

En este capítulo se han desarrollado algoritmos secuenciales para encontrar el tamaño del menor árbol binario en el refinamiento de un símplex que dan respuesta la pregunta 1.3. Se presenta primero una versión recursiva y tras ello otra iterativa con la finalidad de poder estudiar las ventajas e inconvenientes de estos dos enfoques de programación. Debido a la complejidad del problema que se está tratando de resolver, el coste computacional de los algoritmos diseñados es muy alto. En este capítulo se describen estrategias para reducir la búsqueda y por lo tanto, el coste computacional de los algoritmos. El capítulo finaliza presentando los resultados de la ejecución de los algoritmos para la resolución de diferentes instancias del problema en distintas máquinas. Dado el alto requerimiento de memoria de los algoritmos se observa como el uso de diferentes gestores de memoria dinámica afecta en gran medida a los tiempo de ejecución de los algoritmos. La versión recursiva consume un menor tiempo de ejecución para resolver las instancias del problema planteadas.

Capítulo

4

Algoritmos paralelos para encontrar el tamaño del menor árbol binario en el refinamiento de un símplex

La obtención del tamaño del menor árbol binario de que aparece durante el proceso de bisección iterativa de un símplex, es una tarea computacionalmente muy costosa. En este capítulo se van a presentar mecanismos de resolución de este problema haciendo uso de computación de altas prestaciones. Se pretende dar respuesta a la pregunta 1.4: ¿cuál es el modelo de programación paralelo más eficiente para resolver este problema en sistemas multicore de memoria compartida? Para ello, en la sección 4.1 se hace una introducción a la computación paralela. En la sección 4.2 se presentan algoritmos paralelos que hacen uso de un número fijo de hebras de ejecución y en la sección 4.3 se estudian algoritmos paralelos con un número dinámico de hebras. El capítulo finaliza con la sección 4.4 en la que se muestran los resultados de ejecución de los algoritmos vistos en este capítulo y se hace una comparativa entre todos ellos.

4.1 Introducción a la computación paralela

Los algoritmos de optimización pueden requerir una enorme cantidad de cómputo y memoria. Debido a estos grandes requerimientos, se deben utilizar mecanismos que aceleren el proceso de búsqueda y minimicen los requisitos de memoria [64]. El proceso de mejora de los algoritmos de refinamiento puede ir orientado a descubrir nue-

vas técnicas o procedimientos en el ámbito secuencial del algoritmo, o bien, a estudiar nuevos métodos y estrategias que presenten algoritmos paralelos que hagan uso de múltiples procesadores de forma eficiente. En el ámbito secuencial, las mejoras pueden ir encaminadas a diseñar nuevas reglas de eliminación que permitan evaluar un número inferior de elementos o aprovechar las cualidades individuales de un problema de optimización concreto para diseñar nuevos métodos y algoritmos más eficientes. En general, cualquier aspecto del algoritmo de Optimización Global es susceptible de ser mejorado.

En este capítulo se van a diseñar algoritmos paralelos que determinen el menor número de símplices del árbol binario que aparece durante el proceso de refinamiento de un símplex regular. Esta es una tarea con un alto grado de complejidad y por tanto se hace necesario el uso de la computación de altas prestaciones para su resolución en un intervalo de tiempo razonable. La programación paralela y el diseño de programas paralelos eficientes es uno de los pilares básicos de la computación de altas prestaciones.

Para que sea posible la aplicación de computación paralela, la carga computacional ha de ser dividida en diferentes partes o tareas que serán asignadas a diferentes recursos de ejecución que son capaces de trabajar de manera simultanea. Se desea que las tareas sean lo más independientes las unas de las otras y que el algoritmo paralelo desarrollado deba ser capaz de distribuirlas de modo que sea posible realizar una ejecución simultanea en paralelo de ellas.

Para obtener un programa paralelo, el algoritmo ha de ser formulado en un lenguaje de programación adecuado para ello. La ejecución paralela es a menudo controlada por librerías especificas en tiempo de ejecución o directivas del compilador que son añadidas a un lenguaje de programación como C, Fortran o Java.

Los procesadores actuales disponen de varias unidades computacionales en un solo chip que son controladas de manera independiente y que son capaces de acceder a la misma memoria de manera concurrente. El término core o núcleo es utilizado para nombrar a cada una de las unidades computacionales independientes mientras que el término multicore hace referencia a un procesador completo con varios núcleos en su interior. Hoy día, la mayoría de los ordenadores personales son pequeños sistemas paralelos. El auge en el desarrollo tecnológico de procesadores multicore ha sido debido a razones físicas, la velocidad que gobierna la frecuencia de los chips que contienen cada

vez más transistores no se puede seguir incrementando como se había estado haciendo hasta ahora sin dar lugar a un sobrecalentamiento del chip.

Un primer paso en la programación paralela es el diseño especifico de algoritmos o programas paralelos para una aplicación determinada. El diseño comienza con las descomposición de las unidades de computación de la aplicación en varias partes, llamadas tareas, que pueden ser ejecutadas de manera simultanea en los diferentes núcleos del procesador o de la arquitectura paralela. La descomposición de un problema en tareas que puedan ser ejecutadas de manera paralela es un trabajo complicado y laborioso ya que, a menudo, existen diferentes maneras de descomponer una aplicación, cada una de las cuales, tiene sus pros y sus contras. Al tamaño de las tareas, por ejemplo en términos del número de instrucciones que realiza cada una de ellas, se le denomina granularidad. Definir las tareas de manera adecuada es uno de los principales retos en el desarrollo de un algoritmo paralelo y es muy difícil de automatizar. El paralelismo potencial de cada aplicación influye en gran medida en cómo puede ser dividida esa aplicación en tareas independientes. Del mismo modo que existen aplicaciones fácilmente paralelizables debido a la propia independencia de sus tareas, existen otras aplicaciones en las que la fuerte relación y dependencia entre tareas hace que el obtener versiones paralelas eficientes sea un trabajo arduo y complicado.

Las tareas de una aplicación son codificadas en un lenguaje o entorno de programación y son asignadas a procesos o hebras, que a su vez son asignados a unidades físicas de computación para llevará a cabo su ejecución. La asignación de tareas a hebras se denomina planificación y establece el orden de ejecución de las tareas. La planificación se puede realizar a mano en el código fuente o por el entorno de programación y puede realizarse en tiempo de compilación o dinámicamente en tiempo de ejecución. La asignación de hebras a unidades físicas, procesadores o cores, se denomina mapeo y es realizada por el sistema operativo en tiempo de ejecución aunque a veces puede estar influenciada por el programador. Las tareas dentro de un algoritmo paralelo pueden ser independientes, pero también puede surgir la necesidad de establecer un control de dependencias de datos. Si una tarea necesita para su ejecución los datos de otra, las tareas deberán ejecutarse en orden de modo que hasta que los datos de salida de una tarea no estén disponibles, la tarea que los necesita no continúa su ejecución. Por lo tanto, las dependencias de datos crean restricciones en la programación paralela y posibles estados de espera en su ejecución lo que limita su eficiencia. Por lo tanto,

los programas paralelos necesitan de la coordinación y sincronización de hebras para poder ejecutarse correctamente. Los métodos de sincronización y coordinación en computación paralela están fuertemente relacionados con la manera en que la información es intercambiada entre procesos o hebras, y esto depende de la organización de memoria y del hardware.

El concepto de hebra es una generalización del concepto de proceso: un proceso puede consistir en varias hebras que comparten un espacio de direcciones común mientras que cada proceso trabaja en espacios de direcciones diferentes. Cual de estos dos modelos es mas adecuado para una situación dada, dependerá de la organización de la memoria física del entorno de ejecución. Es común distinguir entre dos organizaciones de memoria, máquinas de memoria compartida y máquinas de memoria distribuida. A menudo el término hebra está relacionado con la memoria compartida mientras que el término proceso se asocia con memoria distribuida. Para máquinas de memoria compartida, incluyendo procesadores multicore, existen memorias locales a los procesadores y memorias de acceso común. Los mecanismos de coherencia de las memorias cache locales permite que los datos de la aplicación se encuentren en un estado coherente en las distintas caches. El intercambio de información entre hebras se realiza gracias al uso de variables compartidas que son escritas por una hebra y leídas por otra. El correcto comportamiento de un programa se consigue gracias a al acceso de forma exclusiva en la escritura de un dato compartido. Es decir, una hebra no leerá un dato mientras que la operación de escritura de ese dato por otra hebra no haya finalizado. Para máquinas de memoria distribuida, cada procesador tiene su propia memoria privada y el intercambio de información se realiza mediante el envío de datos de un procesador a otro a través de una red de interconexión utilizando para ello operaciones explícitas de comunicación.

Un aspecto importante de la computación paralela es el tiempo de ejecución del algoritmo paralelo donde intervienen diferentes factores como el tiempo de computación de los procesadores y el tiempo necesario para el intercambio de datos. El tiempo de ejecución del algoritmo paralelo debería reducir el tiempo del algoritmo secuencial en un factor directamente relacionado con el número de cores usados. El tiempo de ejecución paralelo es el tiempo transcurrido desde el comienzo de la aplicación en el primer procesador hasta la finalización de la aplicación en todos y cada uno de los procesadores. Este tiempo se ve afectado por la distribución de los procesadores o núcleos, el

tiempo que se emplea en el intercambio de información o sincronización y los tiempos ociosos en los que el procesador no puede realizar trabajo porque está a la espera de que suceda algún evento. En general, un tiempo paralelo bajo es consecuencia en primer lugar de un buen balanceo de la carga computacional entre las unidades de cómputo, y en segundo lugar de una baja sobrecarga debido a intercambio de información, sincronización y tiempos ociosos de los procesadores. Encontrar una planificación especifica y una estrategia de mapeo que lleve a la obtención de un buen balanceo de carga y una baja sobrecarga en las comunicaciones es a menudo bastante difícil ya que, por ejemplo, reducir la sobrecarga debida al intercambio de información puede dar lugar a un desbalanceo de la carga, y si se desea un buen balanceo de carga a menudo se ha de añadir una sobrecarga mayor debida a la sincronización.

Para realizar una evaluación del rendimiento de los algoritmos paralelos se usan medidas de la ganancia en velocidad y la eficiencia. Estas medidas comparan el tiempo de ejecución paralelo con el tiempo de ejecución secuencial.

El modelo de programación se encuentra en el siguiente nivel de abstracción y describe el sistema de computación paralelo en términos de la semántica de un determinado lenguaje de programación o entorno de programación. Un modelo de programación paralela especifica a través de librerías la visión que van a tener los programadores de la computadora paralela. Por tanto, existen diferentes modelos de programación paralela incluso para la misma arquitectura. Existen varios criterios que ayudan a distinguir entre diferentes modelos de programación:

- El nivel de paralelismo explotado en la ejecución paralela (nivel de instrucción, nivel de bucle, nivel de procedimiento o nivel de programa).
- La especificación del paralelismo implícita o explicita definida por el usuario.
- El modo de ejecución de las unidades paralelas. Actualmente, las más comunes son: Multiple Instruction Multiple Data (MIMD) y Single Program Multiple Data (SPMD).
- Los modos y modelos de comunicación entre las unidades de computación para el intercambio de información (comunicación mediante paso de mensajes o variables compartidas)
- Los mecanismos de sincronización para organizar la computación y la comunicación entre unidades paralelas.

Cada lenguaje o entorno de programación paralelo implementa una combinación de los criterios arriba descritos. Debido a esto existen multitud de combinaciones posibles que dan lugar a múltiples lenguajes o entornos de programación, todos ellos con la finalidad común de proporcionar un mecanismo mediante el cual el programador pueda crear programas paralelos.

Los procesos o hebras que se ejecutan en un programa paralelo pueden ser creados estáticamente al comienzo del programa o pueden ser creados en tiempo de ejecución según las necesidades especificas del programa y de los recursos disponibles. Para el caso estático, un número fijo de procesos o hebras se crean al comienzo del programa. Estos procesos o hebras van a existir durante toda la ejecución del programa paralelo y se destruyen cuando finaliza la ejecución del programa. Un enfoque alternativo es permitir la creación y destrucción de procesos o hebras dinámicamente en puntos arbitrarios de la ejecución del programa. Al comienzo del programa, tan solo un proceso o hebra está activo y ejecuta el programa principal. Durante la ejecución del programa se irán creando y destruyendo diferentes procesos o hebras. El programa finalizará con la destrucción de todas las hebras activas cuando no exista más trabajo por realizar.

Algunos entornos de programación paralela generan una colección de tareas que son ejecutadas por procesos o hebras en múltiples procesadores. Para estructurar un programa paralelo se pueden utilizar diferentes modos de organización que se reflejan en modelos específicos de programación. Estos modelos proporcionan las estructuras específicas de coordinación para los procesos o hebras. Por simplicidad, de aquí en adelante vamos a restringir la atención al uso de hebras, si bien los modelos pueden ser aplicados también para la coordinación de procesos. Si bien existen multitud de modelos específicos de programación paralela, en esta tesis se han utilizado dos, los Posix Threads (PThreads) y los Conjuntos de Tareas, en concreto Intel Threading Building Blocks (TBB).

Posix Threads (PThreads)

Los PThreads definen un estándar para la programación con hebras basado en el lenguaje de programación C. Las hebras de un proceso comparten un mismo espacio de direcciones. Por lo tanto, las variables globales y los objetos de datos generados dinámicamente pueden ser accedidos por todas las hebras de un proceso. Adicionalmente, cada hebra tiene una pila de ejecución independiente que es utilizada para el

control de sus funciones y para almacenar sus variables locales. Estas variables declaradas como locales dentro de las funciones son datos locales de la ejecución de una hebra y por tanto no pueden ser accedidos por cualquier otra hebra.

En el modelo de PThreads, el programador debe particionar el programa en un número adecuado de hebras que puedan ser ejecutadas de manera concurrente unas con otras. Las hebras creadas por el usuario son mapeadas por el planificador de la librería en hebras del sistema que son a su vez mapeadas en los procesadores del sistema por el planificador del sistema operativo. El programador no puede controlar el planificador del sistema operativo y tiene solo una pequeña influencia sobre el planificador de la librería. Por lo tanto, el programador no podría realizar de manera directa el mapeo entre las hebras a nivel de usuario y los procesadores del sistema. Esto facilita el desarrollo de programas, pero también impide el mapeo eficiente que puede realizar el programador de acuerdo a las necesidades especificas de un determinado programa. Es importante resaltar que si bien existen extensiones del sistema operativo que permiten confinar la ejecución de una hebra en un núcleo especifico, en la mayoría de los casos, la planificación proporcionada por la librería y el sistema operativo proporciona buenos resultados y alivia al programador de esfuerzos adicionales de programación.

Conjuntos de Tareas (TBB)

Un programa basado en hebras normalmente tiene que realizar varias operaciones o tareas. Siguiendo una estrategia simple, cada tarea puede ser definida como una función diferente que será llamada por hebras diferentes que ejecutan exactamente esa función y tras ello son destruidas. Dependiendo de la granularidad de las tareas, esto puede llevar a la creación y destrucción de un número muy alto de hebras con la consiguiente sobrecarga computacional asociada. Para muchas aplicaciones, se puede obtener una implementación más eficiente mediante el uso de conjuntos de tareas.

En general, un conjunto de tareas es una estructura de datos en la cual las tareas que se deben realizar son almacenadas y de la cual se pueden extraer tareas para ser llevadas a cabo. Una tarea consta de una serie de operaciones que deben ser ejecutadas y una especificación de los datos sobre los cuales las operaciones deben ser realizadas. A menudo las operaciones a realizar son especificadas como una llamada a una función. Normalmente se utiliza un número fijo de hebras para el procesamiento de las

tareas. Las hebras se crean al comienzo del programa por la hebra principal y se destruyen una vez que todas las tareas hayan sido procesadas. Para las hebras, un conjunto de tareas es una estructura de datos de la cual pueden sacar tareas para procesarlas. Durante el procesamiento de una tarea, una hebra puede generar nuevas tareas e insertarlas en el conjunto de tareas. El acceso a un conjunto de tareas compartido debe ser síncrono para evitar condiciones de carrera, es decir, una corrupción de los datos debido a la ejecución de tareas en un orden incorrecto. La ejecución del programa paralelo finalizará cuando no exista un conjunto de tareas con tareas pendientes y cuando cada hebra haya terminado su última tarea. Los conjuntos de tareas proporcionan un esquema de ejecución flexible que es especialmente útil para aplicaciones adaptativas e irregulares en las cuales las computaciones a llevar a cabo no son fijas al comienzo del programa. Ya que se utiliza un número fijo de hebras, la sobrecarga debido a la creación de hebras suele ser menor que la sobrecarga debida a la creación del número de tareas a ser procesadas. La flexibilidad está asegurada ya que las tareas pueden ser generadas dinámicamente en cualquier punto durante la ejecución del programa. La estructura de datos del conjunto de tareas puede estar proporcionada por el entorno de programación o puede ser incluida por el programador en el programa paralelo. En esta tesis se usará el conjunto de tareas proporcionado por el entorno de programación Threading Building Blocks y se usará un conjunto de tareas definido por el usuario cuando el entorno de programación es PThreads.

Por lo tanto, en esta tesis se van a utilizar los modelos de programación basados en PThreads y TBB para resolver el problema de obtención del número mínimo de símplices que contiene el árbol binario que aparece al realizar el refinamiento de un símplex regular utilizando la técnica de bisección por el lado mayor. Aparentemente no parece excesivamente complicado la aplicación de computación paralela a este tipo de problemas ya que en principio diferentes ramas del árbol pueden ser procesadas de manera independiente en paralelo. Sin embargo, el problema que aparece en la búsqueda del tamaño mínimo del árbol es que diferentes ramas pueden tener diferente profundidad (ver figura 1.7) y por tanto la carga computacional asociada al proceso de cada rama es potencialmente diferente. Esto hace necesario el uso de un balanceo dinámico de la carga para que el algoritmo paralelo sea eficiente.

La ventaja de TBB es que realiza un balanceo dinámico mediante el robo de tareas del conjunto de tareas asociado a otra hebra. El balanceo dinámico de la carga reali-

zado en la versión de Pthreads es inherente a la creación dinámica de hebras como las realizadas en [2, 31, 32, 33, 69, 70].

4.2 Versiones paralelas que utilizan un número fijo de hebras

La computación paralela se puede aplicar al problema de obtención del árbol de menor tamaño mediante el procesado en paralelo de las ramas del árbol de búsqueda. Las diferencias entre las longitudes de las ramas (ver figura 1.7) hacen de este un problema irregular donde el balanceo dinámico de carga es necesario para mejorar el rendimiento del algoritmo paralelo. Las diferencias entre las longitudes de las ramas crecen con la dimensión del problema n.

En los dos algoritmos estudiados en esta sección, el número de hebras ($NThreads$) es estático y es establecido por el usuario. El primero (ver sección 4.2.1) no usa un balanceo dinámico de la carga y se utiliza para describir la dificultad y el desbalanceo del problema. El segundo usa la librería TBB que realiza un balanceo dinámico de carga por *robo de tareas* (ver sección 4.2.2).

4.2.1 Algoritmo Basado en Tres Fases

Este algoritmo tiene tres fases. En la primera fase, el árbol general se visita en modo *hacia delante* (ver descripciones del capítulo 3) usando una búsqueda en anchura hasta que hay suficiente trabajo para cada hebra. La fase 2 consiste en la ejecución del algoritmo R-STS() (ver algoritmo 3.1) para cada uno de los nodos hoja generados en la primera fase. La última fase se encarga de recoger los datos parciales generados en la fase 2 para obtener la solución al problema. Las tres fases se muestran en al algoritmo 4.1.

Fase 1: Lineas 1 a 8. El árbol general se construye parcialmente utilizando búsqueda en anchura (línea 6) para generar nodos en los niveles más altos del árbol. Esta fase acaba cuando el número de nodos hoja ($|\Lambda|$) es mayor o igual al parámetro dado $MinNLN$ (número mínimo de nodos hoja) que es mayor o igual que el número de hebras y cuyo valor se estudiará más adelante en esta sección.

Fase 2: Se crean $NThreads$ hebras en el algoritmo 4.1, línea 10. Cada hebra busca un nodo hoja no procesado N (algoritmo 4.2, línea 1). El algoritmo recursivo R-STS() se ejecuta para cada N y se obtiene el tamaño del árbol menor para ese nodo hoja N. Una hebra finalizará cuando no existan más nodos hoja que procesar. El algoritmo 3S-R-STS (Three Stages - Recursive - Smallest Tree Size, denotado por simplicidad como T3S, Tree 3 Steps) espera a la finalización de todas las hebras en la línea 11 antes de ejecutar la siguiente fase.

Fase 3: El algoritmo 4.3 se ejecuta en el nodo raíz R del árbol completo (algoritmo 4.1, línea 12) para recoger los resultados obtenidos durante la fase 2 y determinar el tamaño del árbol más pequeño. El árbol completo, hasta los nodos hoja de la fase 2, se recorre mediante el uso de llamadas recursivas (línea 3). El tamaño del árbol más pequeño (STS) del nodo actual se calcula en la línea 4. El algoritmo finaliza cuando retorna de la recursión al nodo raíz devolviendo el valor de STS en el nodo raíz.

La figura 4.1 muestra las tres fases del algoritmo 4.1 para un 3-símplex cuando $NThreads$=4 y $MinNLN$=8. Los círculos representan los nodos y los cuadrados las diferentes opciones de selección de LM. La primera fase finaliza cuando el número de nodos hoja es ocho (dos en nivel 3 y seis en nivel 4). En la fase dos, las cuatro hebras ejecutan el algoritmo R-STS() en cada nodo hoja para calcular su STS. El número de nodos hoja procesados por cada hebra puede ser diferente y depende del trabajo computacional de cada nodo hoja. Finalmente, en la fase tres, se recoge la información obtenida desde los nodos hoja hasta el nodo raíz para devolver el valor STS del nodo raíz como resultado.

La principal dificultad del algoritmo T3S es determinar un buen valor de $MinNLN$. Asignar el valor $NThreads$ a $MinNLN$ no es buena idea porque obtener un número de nodos hoja en la primera fase igual al número de hebras $NThreads$ no siempre es posible debido a los diferentes factores de ramificación que puede tener cada nodo. Por otro lado, $|\Lambda| > MinNLN$=$NThreads$ normalmente produce hebras que evalúan un número diferente de nodos y por lo tanto, provocará desbalanceo. Incluso cuando es posible tener $|\Lambda|$=$MinNLN$ = $NThreads$ el desbalanceo aparecerá debido a los diferentes tamaños de los subárboles.

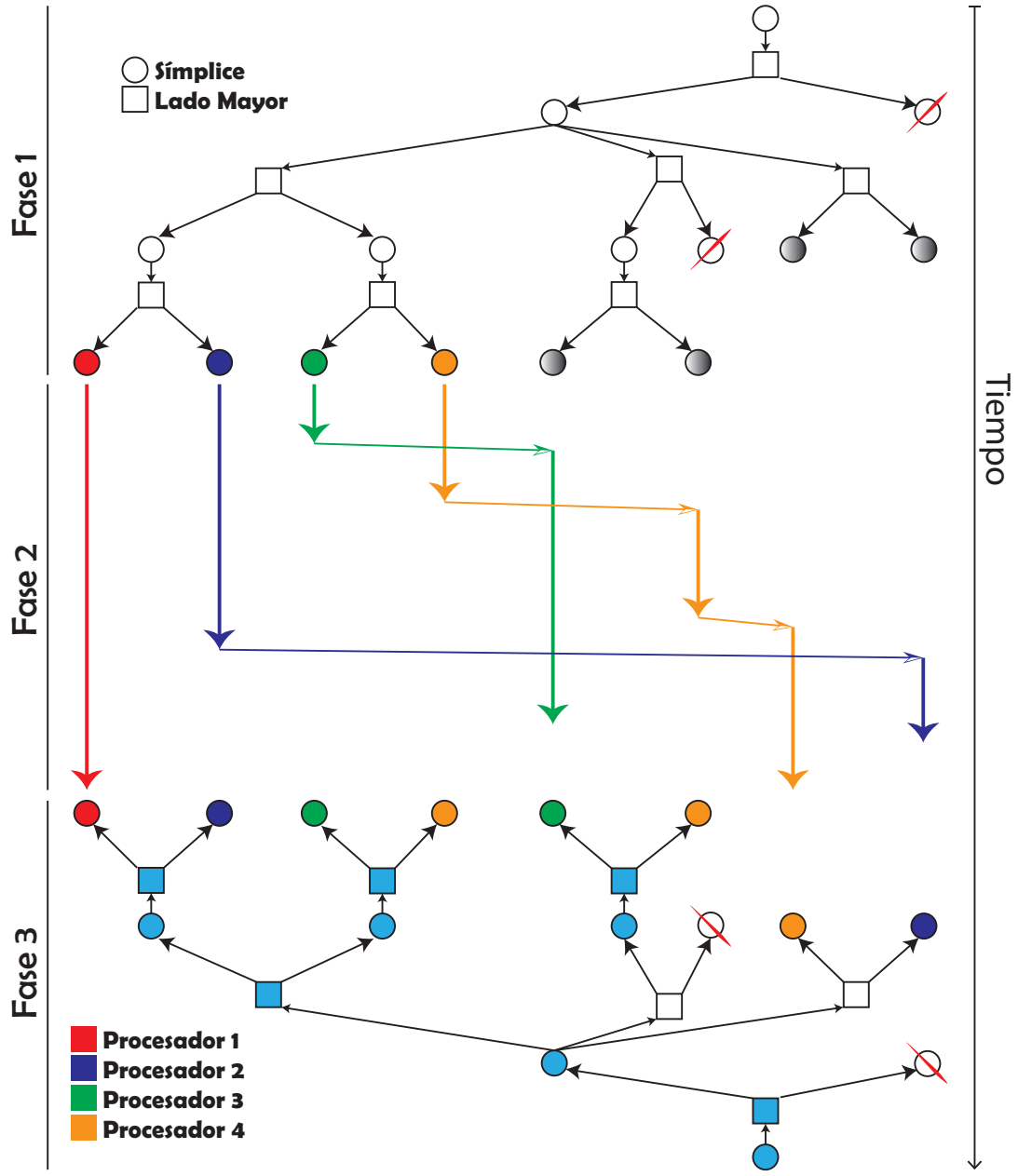


Figura 4.1: Ejemplo del algoritmo 4.1 para un 3-símplex cuando $NThread$=4 y $MinNLN$=8

Algoritmo 4.1 3S-R-STS(S_1, ϵ, $NThreads$, $MinNLN$)

▶ Denotado como T3S (Tree 3 Steps) por simplicidad

Requiere:

S_1: símplex inicial

ϵ: precisión

$NThreads$: número de hebras

$MinNLN$: número mínimo de nodos hoja

1: **if** $\omega(S_1) \leq \epsilon$ **then**
2: **return** 1
3: R:= CreaNodo(S_1, NULL, 0) ▶ Ver algoritmo 3.2
4: $R \rightarrow \Lambda$ ▶ Λ: Conjunto de hojas del árbol general
5: **while** $|\Lambda| <$ MinNLN **do** ▶ Número insuficiente de nodos hoja
6: $N \leftarrow \Lambda$ ▶ Continuar por el nodo hoja más antiguo (búsqueda en anchura)
7: CreaNodosHijo(N) ▶ Ver algoritmo 3.3
8: $\{CN_j \in N$ con $CSTS_j = 0\} \rightarrow \Lambda$ ▶ Los nodos hijos (CN) ahora son hojas
9: **for** $i \leftarrow 0, NThreads$ **do**
10: Crea Thread-3S-R-STS(Λ, ϵ) ▶ procesa hojas en paralelo
11: Esperar a todas las hebras
12: **return** Gather-3S-R-STS(R) ▶ Recoger resultados del nodo raiz

Algoritmo 4.2 Thread-3S-R-STS(Λ, ϵ)

Requiere: Λ: conjunto de nodos hoja, ϵ: precisión

1: **while** $\exists N \in \Lambda$ con STS=0 **do** ▶ Acceso exclusivo a N
2: $STS \in N$:= R-STS ($S \in N$, ϵ) ▶ Ver algoritmo 3.1

La tabla 4.1 muestra algunas estadísticas del algoritmo 4.1 para un 3-símplex cuando $\epsilon = 0,025$ y $MinNLN$ = $NThreads$, para diferentes valores de $NThreads$. La tabla 4.2 contiene información similar, pero referida a un 4-símplex cuando $\epsilon = 0,125$. Las columnas muestran la siguiente información:

- $|\Lambda|$: Número de nodos hoja al finalizar la primera fase.
- $Av.NN$: Número medio de nodos evaluados por hebra. Si NN_i es el número de nodos evaluados por una hebra i entonces $Av.NN = \sum_i NN_i/NThreads$.
- $Av.Imb$: Desbalanceo medio relativo a $Av.NN$. Se define el desbalanceo de la hebra i relativo a $Av.NN$ como $Imb_i = |NN_i - Av.NN|/Av.NN$. Entonces, $Av.Imb = \sum_i Imb_i/NThreads$.

Algoritmo 4.3 Gather-3S-R-STS(N)

Requiere: N: un nodo del árbol general.

1: **if** $STS \in N = 0$ **then**
2: **while** $\exists\, CSTS_j = 0$ **do**
3: $CSTS_j \in N$:= Gather-3S-R-STS(CN_j) ▶ Llamada recursiva
4: STS:=mín$_k\{CSTS_k + CSTS_{k+1}\} + 1,\ k \mod 2 = 1$
5: **return** STS

- $Max.Dev$: Desviación máxima relativa a $Av.NN$.
  $Max.Dev = \max_i |NN_i - Av.NN|/Av.NN$.

Tabla 4.1: Estadísticas del algoritmo 4.1 para un 3-símplex cuando $\epsilon = 0,025$ y $MinNLN$ = $NThreads$

$NThreads$	$\|\Lambda\|$	$Av.NN$	$Av.Imb$	$Max.Dev$
1	1	2,25·10^7	0,000	0,000
2	5	1,12·10^7	0,167	0,000
4	5	5,63·10^6	0,204	0,204
8	8	2,82·10^6	0,402	0,217
16	17	1,41·10^6	0,469	1,613
32	32	703.963	0,644	0,767
64	64	531.981	0,370	0,359
128	128	175.990	0,466	1,065
256	258	87.994	0,508	3,207

Las tablas 4.1 y 4.2 muestran los resultados cuando $|\Lambda| \geq MinNLN = NThreads$, para un 3-símplex y un 4-símplex, respectivamente. Por ejemplo, en la tabla 4.1, para $NThreads$=8, el desbalanceo medio es del 40 %, con una desviación máxima del 22 %, mostrando la irregularidad del problema y las grandes diferencias entre los tamaños de los subárboles. La tabla 4.2 muestra tendencias similares para un 4-símplex. Aumentar el valor de $MinNLN$ por encima del valor de $NThreads$ produciría una mejor distribución de los nodos hoja entre las hebras, reduciendo de este modo el desbalanceo, pero aumentando el tiempo dedicado a la fase 1, que se realiza en secuencial. $MinNLN$ debe tener en cuenta el tamaño del problema n, la precisión ϵ requerida y

Tabla 4.2: Estadísticas del algoritmo 4.1 para un 4-símplex cuando $\epsilon = 0,125$ y $MinNLN = NThreads$

$NThreads$	$\|\Lambda\|$	$Av.NN$	$Av.Imb$	$Max.Dev$
1	1	1,82·10^8	0,000	0,000
2	9	9,09·10^7	0,098	0,000
4	9	4,54·10^7	0,115	0,115
8	9	2,27·10^7	0,181	0,544
16	17	1,14·10^7	0,706	0,278
32	33	5,68·10^6	0,569	2,340
64	64	2,84·10^6	0,364	0,632
128	132	1,42·10^6	0,504	1,193
256	257	710.032	0,766	2,419

el número de hebras $NThreads$. Adicionalmente, se desea que $|\Lambda|$ tenga un valor cercano a un múltiplo de $NThreads$, lo cual es una heurística que no tiene porque ser ventajosa en todos los casos. Se proponen las siguientes heurísticas:

Mejora 4.1. $MinNLN = n \cdot NThreads/8\epsilon$.

Mejora 4.2. *Cambiar la línea 5 del algoritmo 4.1 a:*

3. **While not** *(* $|\Lambda| \geq MinNLN$ **y**
($|\Lambda| \mod NTrheads = 0$ **o** $|\Lambda| \mod NTrheads \geq 0{,}8 \cdot NThreads$*))* **do**

Las tablas 4.3 y 4.4 proporcionan la misma información que las tablas 4.1 y 4.2, respectivamente, pero utilizando las mejoras 4.1 y 4.2.

El desbalanceo medio y la desviación media son ahora bajos para valores relativamente altos de $MinNLN$, produciendo una buena distribución de los nodos hoja entre las distintas hebras. Aun así, también se incrementa la carga computacional de las partes secuenciales del algoritmo, las fases 1 y 3. Estos resultados muestran la necesidad del uso del balanceo dinámico de carga.

4.2.2 Algoritmo Intel Threading Building Blocks

Uno de los objetivos de TBB es reducir significativamente el problema del desbalanceo de carga gracias a la técnica de *robo de tareas* [6]. El entorno de programación

Tabla 4.3: Estadísticas del algoritmo 4.1 para un 3-símplex cuando $\epsilon = 0{,}025$ y con las mejoras 4.1 y 4.2

$NThread$	$\lvert\Lambda\rvert$	$Av.NN$	$Av.Imb$	$Max.Dev$
1	8	$2{,}25{\cdot}10^7$	0,00	0,00
2	30	$1{,}12{\cdot}10^7$	0,00	0,00
4	64	$5{,}63{\cdot}10^7$	0,02	0,02
8	464	$2{,}82{\cdot}10^7$	0,02	0,02
16	1.844	$1{,}41{\cdot}10^7$	0,01	0,01
32	7.373	703.755	0,01	0,01
64	16.384	351.746	0,01	0,01
128	117.967	175.177	0,02	0,02
256	471.862	86.323	0,01	0,01

TBB sugiere a los programadores expresar la concurrencia en términos de *tareas* en lugar de hebras. Las tareas son regiones especiales de código que realizan una función específica cuando son ejecutadas por la librería de ejecución de TBB. Las tareas tienen una característica muy importante que puede hacer que el programador se decante por su uso en vez de usar las hebras: las tareas tienen normalmente unos tiempos de sobrecarga mucho menores debido a que pueden ser creadas y destruidas de una manera más eficiente que las hebras. Por ejemplo, si se utiliza el sistema operativo Linux, las tareas tienen unos tiempos de creación y finalización hasta 18 veces más rápidos que las hebras. Utilizando sistemas Windows esta mejora es de hasta más de 100 veces más rápida [65].

TBB puede verse como una librería para la resolución de algoritmos paralelos en la que el programador puede recurrir a modelos de alto nivel que le facilitan la implementación de un `for_paralelo`, reducciones paralelas o algoritmos pipeline, pero también es posible trabajar a un nivel más bajo directamente utilizando la clase tarea. De hecho, para un algoritmo basado en árboles, como el que se trata en esta tesis, la implementación basada en este tipo de tareas de bajo nivel es especialmente adecuada. De este modo, una tarea será la encargada de procesar cada nodo del árbol y, si es necesario, crear nuevas tareas para cada uno de los nodos hijos que aparecen en las divisiones de los lados mayores. TBB tiene dos características fundamentales a la hora de realizar recorridos de árboles en paralelo, la primera es que pretende equilibrar los

Tabla 4.4: Estadísticas del algoritmo 4.1 para un 4-símplex cuando $\epsilon = 0{,}125$ y con las mejoras 4.1 y 4.2

$NThread$	$\|\Lambda\|$	$Av.NN$	$Av.Imb$	$Max.Dev$
1	8	1,82·10^8	0,00	0,00
2	30	9,08·10^7	0,03	0,00
4	64	4,54·10^7	0,02	0,08
8	464	2,27·10^7	0,08	0,01
16	1.844	1,13·10^7	0,06	0,09
32	7.373	2,83·10^6	0,07	0,01
64	16.384	1,42·10^6	0,08	0,02
128	117.967	709.278	0,06	0,01
256	471.862	352.289	0,04	0,02

recorridos del árbol en anchura y en profundidad (el primero asegura el paralelismo mientras que el segundo evita un consumo excesivo de memoria). La segunda es que, gracias a la técnica del robo de tareas, TBB realiza un balanceo dinámico de la carga con una sobrecarga computacional relativamente baja. Las dos características mencionadas, son particularmente relevantes en algoritmos irregulares de Ramificación y Acotación.

TBB genera rápidamente suficientes tareas al comienzo del recorrido del árbol para poder utilizar completamente las capacidades multinúcleo del sistema, tan pronto como sea posible. Esto es posible gracias al planificador con capacidad de robo de tareas que cada hebra activa posee en ejecución: la tarea principal crea tareas adicionales que son robadas rápidamente por el resto de hebras. De este modo, siguiendo una estrategia de recorrido en anchura, todas las hebras estarán ocupadas en $log_a b$ pasos, donde a es el número de nodos hijos creados por cada tarea y b es el número total de hebras.

Algoritmo 4.4 TBB-STS(S_1, ϵ, $NThreads$)

► Denotado como TBB (Threading Building Blocks) por simplicidad

Requiere: S_1: símplex inicial, ϵ: precisión, $NThreads$: número de hebras.

1: Iniciar el planificador TBB con $NThreads$ hebras trabajando.
2: a = newRootTask TBB-R-STS(S, ϵ)
3: spawn_root_and_wait(a)

El algoritmo 4.4, denominado TBB-STS (Threading Building Blocks - Smallest Tree Size) pero que por simplicidad llamaremos TBB, muestra la función principal codificada en TBB que tan solo tiene que iniciar el planificador TBB, especificando el número de hebras/núcleos que van a ser utilizados y después de esto lanza la tarea raíz. El código para esta tarea se describe en el algoritmo 4.5, denominado TBB-R-STS (Threading Building Blocks - Recursive - Smallest Tree Size). El algoritmo 4.5 básicamente reproduce el funcionamiento del algoritmo secuencial 3.1 de la Sección 3.1. La única diferencia es que el procesado de los diferentes nodos del árbol general se realiza a través de la generación de nuevas tareas.

Algoritmo 4.5 TBB-R-STS (S_i, ϵ)

1: **if** $w(S_i) \leq \epsilon$ **then**
2: **return** 1
3: **for** cada lado mayor LE_h de S_i **do**
4: $\{S_{2i}, S_{2i+1}\}$:=Bisect(S_i,LE_h)
5: $r_{2i,h}$:= spawn (newTask TBB-R-STS (S_{2i}, ϵ))
6: $r_{2i+1,h}$:= spawn (newtask TBB-R-STS(S_{2i+1}, ϵ))
7: $r_h = r_{2i,h} + r_{2i+1,h}$
8: **return** $1 + \min_h r_h$

Adicionalmente, para evitar que existan tareas esperando a la finalización de sus tareas hijas, se han implementado las siguientes optimizaciones [44].

Mejora 4.3. *TBB permite crear una tarea especial llamada tarea de continuación antes de crear nuevas tareas hijas. La tarea de continuación será la tarea que se ejecutará cuando todas las tareas hijas finalicen. En este estudio, la tarea de continuación ejecuta las líneas 7 y 8 del algoritmo 4.5.*

Mejora 4.4. *TBB también permite el reciclado de tareas. Una tarea se recicla como la última tarea hija, en lugar de crear una nueva tarea para ello. Por lo tanto, el último nodo a procesar, en lugar de ser procesado como una tarea nueva es ejecutado de manera inmediata por la misma hebra que ha creado los otros nodos hijo.*

En algunos problemas como el estudiado aquí, la gestión de un gran número de tareas de grano fino puede suponer una alta sobrecarga en el sistema [74]. La sobrecarga producida por la creación, ejecución, sincronización y destrucción de tareas en TBB puede hacer que el algoritmo paralelo sea menos eficiente que su versión secuencial.

Mejora 4.5. *Establecer un cut-off adecuado puede mejorar el rendimiento limitando la profundidad del algoritmo paralelo. Dicho cut-off determina el punto a partir del cual las operaciones de procesado de nodos han de realizarse de modo secuencial, utilizando el algoritmo R-STS (algoritmo 3.1).*

En la sección 4.4 los resultados para el algoritmo TBB se han obtenido teniendo en cuenta todas las mejoras descritas anteriormente.

4.3 Versiones paralelas que utilizan un número dinámico de hebras

En las versiones que se presentan en esta sección, el usuario establece el número máximo de hebras de los algoritmos paralelos. Por simplicidad, vamos a llamar $NThreads$ al número máximo de hebras (usando la misma nomenclatura que para el número de hebras estáticas utilizadas en la sección anterior) y vamos a denotar mediante $RThreads$ al número de hebras que se encuentran en ejecución. Una hebra nueva se crea cuando $RThreads$ es menor que $NThreads$ y la hebra creadora tiene suficiente trabajo. En caso de que se cree una nueva hebra, el valor de $RThreads$ se incrementa en uno. La asignación de las hebras a los procesadores se realiza por el sistema operativo y esta fuera del ámbito de este trabajo. Cuando una hebra finaliza su trabajo, decrementa el valor de $RThreads$ en uno, devuelve el resultado y muere, permitiendo de este modo la creación de una nueva hebra. Cuando el nivel de paralelismo lo permite, el valor de $RThreads$ será igual al de $NThreads$ durante la mayor parte del tiempo de cómputo [2]. En trabajos futuros se estudiarán algoritmos en los que no sea necesario establecer el parámetro $NThreads$. Las versiones paralelas serán auto-ajustables y podrán adaptar el valor de $RThreads$ a la eficiencia y recursos del sistema en tiempo de ejecución [14, 15, 16, 70].

El algoritmo 4.6, denotado como DNT-STS (Dynamic number of Threads - Smallest Tree Size), será la función principal para las versiones paralelas que utilizan un número dinámico de hebras. Dependiendo del tipo de cómputo realizado por las hebras, se han estudiado dos tipos de algoritmos que se describen en las siguientes secciones.

Algoritmo 4.6 DNT-STS (S_1, ϵ, $NThreads$, *Método)*

Requiere:

S_1: símplex inicial

ϵ: precisión

$NThreads$: número máximo de hebras

Método: Recursivo or Iterativo.

```
1:  RThreads:=1                                   ▶ Número de hebras en ejecución
2:  if Método = Recursivo then
3:      Resultado:=CreaHebra(DNT-R-STS(S1, ϵ))    ▶ Ver algoritmo 4.7
4:  else                                          ▶ Método=Iterativo
5:      if ω(S1) ≤ ϵ then
6:          return 1
7:      N:= CreaNodo(S1, NULL, 0)                 ▶ see algoritmo ??
8:      Resultado:=CreaHebra(DNT-I-STS(N, ϵ))     ▶ Ver algoritmo 4.8
9:  Espera la finalización de las hebras
10: return Resultado
```

4.3.1 Algoritmo paralelo con un número dinámico de hebras que realizan un procedimiento recursivo

El algoritmo 4.7, denotado como DNT-R-STS (Dynamic number of Threads - Recursive-Smallest Tree Size) y que por simplicidad llamaremos RPTH, utiliza una estrategia similar a la de TBB (algoritmo 4.5), pero ha sido codificado usando PThreads por lo que no hace uso de tareas.

El algoritmo 4.7 es una extensión paralela del algoritmo 3.1, donde se crean nuevas hebras para procesar diferentes subárboles. Cada hebra ejecuta el algoritmo secuencial recursivo con la opción de crear nuevas hebras para procesar trabajo pendiente (ver líneas 7 y 12). Como la variable $RThreads$ es compartida por todas las hebras, la primera hebra que lea un valor de $RThreads$ menor que $NThreads$ será la encargada de crear una nueva hebra para que trabaje sobre un nodo (y por lo tanto, un subárbol) pendiente de evaluar. De este modo, el balanceo dinámico de carga es inherente a la creación dinámica de hebras. Una hebra cuando finaliza su trabajo decrementa el valor de $RThreads$ (línea 16), devuelve su resultado (línea 17) y muere. Una hebra realiza una llamada recursiva cuando no puede crear nuevas hebras o cuando está procesando el último lado mayor del símplex (ver líneas 9 y 14). Esto es muy parecido al *reciclado*

Algoritmo 4.7 DNT-R-STS (S_i, ϵ)

▶ Denotado como RPTH (Recursive PThreads) por simplicidad.

Requiere: S: símplex, ϵ: precisión.

```
1:  if w(S_i) ≤ ε then
2:      return 1
3:  for cada lado mayor LE_h de S_i do
4:      {S_2i, S_2i+1}:=Bisect(S_i,LE_h)
5:      if RThreads < NThreads then          ▶ RThreads es una variable compartida
6:          RThreads++;
7:          r_2i,h := CreaHebra(DNT-R-STS (S_2i, ε))
8:      else
9:          r_2i,h := DNT-R-STS(S_2i, ε)
10:     if RThreads < NThreads and LE_h no es el último LE then
11:         RThreads++;
12:         r_2i+1,h := CreaHebra(DNT-R-STS(S_2i+1, ε))
13:     else
14:         r_2i+1,h: = DNT-R-STS(S_2i+1, ε)
15: Esperar las hebras
16: RThreads := RThreads -1                  ▶ Esto permite la creación de hebras
17: return 1 + mín_h r_h
```

de tareas que se utiliza en TBB (ver la mejora 4.4). El principal inconveniente del algoritmo es la espera a las hebras creadas que se realiza en la línea 15, porque la hebra actual no puede continuar en el modo *hacia atrás* del recorrido del árbol hasta que obtenga el resultado de sus hebras descendientes.

Una hebra realiza una búsqueda en profundidad, pero la creación de hebras proporciona cierto nivel de búsqueda en anchura. La diferencia con la versión de TBB es que la búsqueda en anchura se realiza normalmente a más profundidad del árbol general, consumiendo por lo tanto, menos memoria.

Como TBB, el algoritmo RPTH puede realizar un *Cut-off* (ver la mejora 4.5) para evitar la generación de nuevas hebras cuando el subárbol a procesar es suficientemente pequeño.

4.3.2 Algoritmo paralelo con un número dinámico de hebras que realizan un procedimiento iterativo

En esta sección se estudia la extensión paralela del algoritmo secuencial iterativo (ver algoritmo 3.4). El objetivo principal es evitar los problemas que aparecían en el algoritmo 4.7. Se pretende que una hebra no tenga que esperar los resultados de las hebras que ha creado para que pueda seguir buscando trabajo. Para ello, la hebra que devuelve un resultado a un nodo será la que compruebe si el trabajo en ese nodo ha finalizado. Esa hebra puede ser diferente de la hebra que creó el nodo. Además, las hebras siempre están buscando trabajo recorriendo el árbol general en el modo *hacia atrás*. En lugar de utilizar una implementación recursiva se ha utilizado una implementación iterativa que facilita que las hebras recorran el árbol general, ya que una hebra podrá moverse a subárboles que no tienen porque ser el propio donde fue creada gracias a los punteros existentes entre nodos del árbol. La idea general es que una hebra se mueva al nodo padre en busca de trabajo cuando en el nodo actual no exista trabajo pendiente que no haya sido asignado a otra hebra. Una hebra que alcance el nodo raiz sin haber encontrado trabajo que realizar muere, permitiendo la creación de una nueva hebra. De este modo la búsqueda de trabajo se realiza desde niveles cercanos a las hojas hacia la raíz del árbol, reduciendo de este modo el consumo de memoria de la parte de búsqueda en anchura realizada cuando se crea una nueva hebra o una hebra existente encuentra nuevo trabajo. Por lo tanto, el algoritmo usa una estrategia de búsqueda híbrida en el que cada hebra realiza una búsqueda en profundidad y la búsqueda en anchura aparece cuando se puede realizar en paralelo el procesamiento de varios hijos de un nodo.

El algoritmo 4.8, denotado como DNT-I -STS (Dynamic number of Threads - Iterative - Smallest Tree Size) y que por simplicidad llamaremos IPTH, muestra el procedimiento llevado a cabo por una hebra. Una hebra ejecuta un bucle infinito (línea 1) hasta que no puede encontrar trabajo que realizar (línea 19) o hasta que calcula el resultado final (línea 30). Como en el algoritmo 3.4, el resultado de un nodo actualiza el elemento correspondiente de $CSTS$ en el nodo padre (línea 26). En la línea 3, se crean los nodos hijo del nodo asignado a una nueva hebra (línea 7). Solo se crean nuevas hebras cuando el número de hebras en ejecución es menor que el máximo y existe un nodo hijo a procesar que no es el último pendiente de ser procesado (ver línea 5).

Algoritmo 4.8 DNT-I-STS (N, ϵ)

▶ Denotado como IPTH (Iterative PThreads) por simplicidad

Requiere: N: un nodo del árbol general, ϵ: precisión.

```
1:  while true do
2:    if CC ∈ N = false then                          ▶ Nodos hijo aun no han sido creados
3:      CreaNodosHijo(N)                                                  ▶ ver algoritmo 3.3
4:    j:= PróximoHijoAProcesar(N, Rem, Finish)                          ▶ ver algoritmo 4.9
5:    while RThreads < NThreads and j ≥ 0 and Rem ≥ 1 do
6:      RThreads := RThreads+1
7:      NewThread(DNT-I-STS(CN_j, ε))          ▶ Asignación de trabajo a una nueva hebra
8:      j := PróximoHijoAProcesar(N,Rem, Finish)
9:    if j = −1 then                              ▶ CSTS ∈ N está bloqueado por otra hebra
10:     N := PN ∈ N                                          ▶ Búsqueda de trabajo en el nodo padre
11:     continue
12:   if j ≥ 0 then      ▶ No se puede crear nueva hebra o solo hay un nodo hijo pendiente
13:     N := CN_j                                          ▶ Esta hebra esta a cargo del nodo hijo
14:     continue
15:   else          ▶ Todos los tamaños de subárboles han sido o están siendo calculados
16:     if !Finish then                ▶ Un nodo hijo esta siendo procesado por otra hebra
17:       if S_i ∈ N = S_1 then                                              ▶ Soy el nodo raíz
18:         RThreads:=RThreads-1                                ▶ Permite la creación de hebras
19:         Esta hebra finaliza                                 ▶ No hay trabajo hacia arriba
20:       else
21:         N := PN ∈ N                                        ▶ Búsqueda de trabajo en el nodo padre
22:         continue
23:     else                      ▶ Todos los tamaños de los subárboles han sido calculados
24:       STS:=mín_k{CSTS_k + CSTS_{k+1}} + 1, k mód 2 = 1
25:       if S_i ∈ N ≠ S_1 then
26:         CSTS_{IinPN} ∈ PN := STS                          ▶ Actualizar CSTS del padre
27:         N:=PN ∈ N   ▶ Buscar trabajo en el nodo padre y liberar el antiguo nodo N
28:         continue
29:       else
30:         return STS                                          ▶ El procedimiento finaliza
```

Mediante una llamada al algoritmo 4.9 (líneas 4 y 8) se obtiene el siguiente hijo a procesar, el número de hijos pendientes de ser procesados y si todos los hijos de un nodo

Algoritmo 4.9 PróximoHijoAProcesar(N, Rem, $Finish$)

Requiere:

N: un nodo del árbol general

Rem: devuelve el número de elementos pendientes de ser procesados CN

$Finished$: ¿han sido todos los CN elementos procesados?

```
1:  if !TryLock para intentar acceso a CSTS ∈ N then
2:      return −1                          ▶ El nodo está bloqueado por otra hebra
3:  Ω := {i : CSTS_i = 0}   ▶ Conjunto de índices de los hijos pendientes de ser procesados
4:  if Ω ≠ ∅ then
5:      Rem := |Ω| − 1
6:      CSTS_Ω0 := −1                      ▶ CN_Ω0 estará en proceso
7:      Desbloquear acceso a CSTS
8:      return Ω0                  ▶ Índice del primer elemento pendiente de CN
9:  if ∃ i : CSTS_i = −1 then              ▶ Existe un nodo hijo siendo procesado
10:     Finish:= false
11: else
12:     Finish:= true
13: desbloquear acceso a CSTS
14: return −2
```

ya han sido procesados. Ya que varias hebras pueden intentar acceder a un nodo en busca de trabajo, la gestión de los elementos de $CSTS$ (y por lo tanto, CN) se lleva a cabo mediante acceso en exclusión mutua. (ver líneas 1, 7 y 13 del algoritmo 4.9). Una hebra no pierde tiempo esperando a que $CSTS$ sea desbloqueado y cuando encuentra un nodo bloqueado la hebra se desplaza al nodo padre en búsqueda de trabajo (ver algoritmo 4.8 línea 10). En caso de que exista un hijo a procesar, pero no sea posible la creación de nuevas hebras, la hebra se encarga de su procesamiento (ver algoritmo 4.8 línea 13). Cuando ya no hay más hijos que procesar, puede ocurrir que el nodo este a la espera de algún resultado para que se pueda finalizar su cómputo. En ese caso, la hebra se desplazará al nodo padre en búsqueda de trabajo (ver línea 21) siempre y cuando no se encuentre en el nodo raíz (ver línea 19). Finalmente, la hebra devuelve el resultado de un nodo finalizado (línea 26) y se mueve al nodo padre (línea 27) o devuelve el resultado final del nodo raíz (ver línea 30). Como en todos los algoritmos paralelos vistos previamente, en este algoritmo se han aplicado las estrategias de reducción de la búsqueda presentadas en la sección 3.3 así como el uso del *Cut-off* (ver la mejora 4.5).

4.4 Resultados

Los algoritmos se han ejecutado en las máquinas BullX y Aloe cuyas características fueron descritas en la sección 3.4.

TBB genera rápidamente suficientes tareas para alimentar todas las hebras, gracias al planificador que toda hebra ejecuta y que realiza el robo de tareas (búsqueda en anchura). Aun así, el subárbol de un nodo (tarea) se procesa mediante una búsqueda en profundidad para evitar la explosión del número de tareas. Aun así, la creación de una tarea por nodo puede ser más costoso que la evaluación secuencial de un subárbol pequeño. Por este motivo se aplica un Cutoff en la búsqueda paralela (ver Mejora 4.5).

En el caso de uso de Pthreads, también se usa un *Cut-off* para forzar que cada hebra realice el procesado de subárboles suficientemente pequeños. Con ello se evita el coste de crear y destruir hebras cuando el subárbol a evaluar es pequeño. Se han probado diferentes valores de *Cut-off* de manera experimental.
Los resultados se han obtenido para las siguientes instancias del problema:

- I3: 3-símplex con $\epsilon = 0{,}025$ y *Cut-off* = $\omega(S_i) \leq 2\epsilon$.
- I4: 4-símplex con $\epsilon = 0{,}125$ y *Cut-off* = $\omega(S_i) \leq 2\epsilon$.
- I5: 5-símplex con $\epsilon = 0{,}25$ y *Cut-off* = $\omega(S_i) \leq 2\epsilon$.
- I6: 6-símplex con $\epsilon = 0{,}5$ y *Cut-off* = $\omega(S_i) \leq 1{,}5\epsilon$.

El valor de *Cut-off* para I6 se ha establecido a $\omega(S_i) \leq 1{,}5\epsilon$ en lugar de $\omega(S_i) \leq 2\epsilon$ porque un valor de *Cut-off* a $\omega(S_i) \leq 2\epsilon$ cuando $\epsilon = 0{,}5$ equivale a no utilizar *Cut-off*.

Los diferentes gestores de memoria descritos en la en la sección 3.4 también han sido probados con los algoritmos 4.1, 4.4, 4.7 y 4.8. El gestor de memoria Lockless Allocator ha demostrado ser la mejor opción en todas las pruebas realizadas. Por lo tanto, de ahora en adelante se va a utilizar solamente Lockless Allocator.

Para comparar los algoritmos paralelos se ha usado la medida de speedup. Se define la ganancia de velocidad o speedup Sp, como la relación entre el tiempo de ejecución del mejor algoritmo secuencial, t_{sec} y el tiempo de ejecución del algoritmo paralelo con p elementos de proceso, $t_{par}(p)$.

$$Sp = \frac{t_{sec}}{t_{par}(p)} \tag{4.1}$$

Para todas pruebas realizadas, el tiempo de ejecución del mejor algoritmo secuencial, t_{sec} ha sido obtenido con el algoritmo 3.1.

4.4.1 Resultados para las instancias I3 e I4 en BullX

Las siguientes tablas muestran el tiempo de ejecución en segundos ($T(s)$) y el speedup (Sp) de los algoritmos paralelos estudiados, para resolver las instancias I3 e I4 en la máquina BullX. Los mejores resultados de speedup se han sombreado.

Las tablas 4.5 y 4.6 muestran los resultados para las instancias I3 e I4, respectivamente. La versión paralela TBB es la mejor en la mayoría de los casos para ambas instancias, seguida de la versión IPTH. Las versiones RPTH y T3S tienen un speedup bastante pobre. La figura 4.2 presenta una gráfica de los speedup de las tablas anterio-

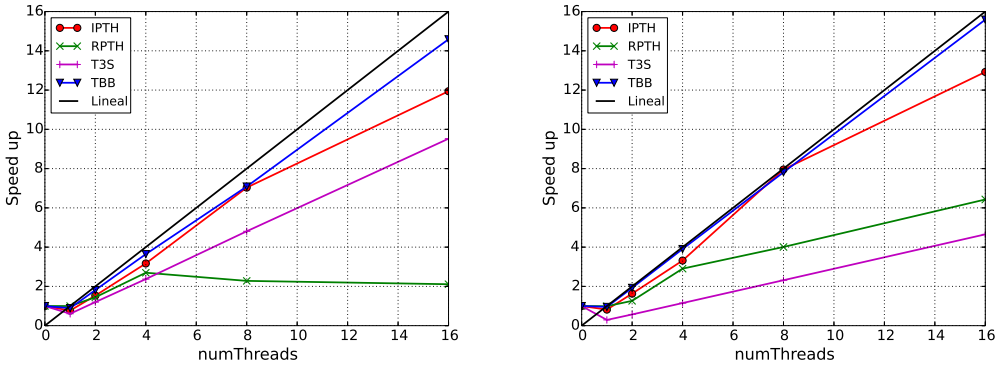
Tabla 4.5: Resultados en Bullx para la instancia I3. $t_{sec} = 30{,}36$ s.

NTh	T3S T(s)	T3S Sp	TBB T(s)	TBB Sp	RPTH T(s)	RPTH Sp	IPTH T(s)	IPTH Sp
1	50,12	0,60	34,06	0,89	30,36	1,00	39,55	0,76
2	25,29	1,20	16,74	1,81	21,00	1,45	19,82	1,53
4	12,79	2,37	8,33	3,64	11,26	2,70	9,58	3,16
8	6,31	4,80	4,28	7,08	13,31	2,28	4,32	7,03
16	3,18	9,52	2,08	14,58	14,39	2,11	2,54	11,94

Tabla 4.6: Resultados en Bullx para la instancia I4. $t_{sec} = 152{,}71$ s.

NTh	T3S T(s)	T3S Sp	TBB T(s)	TBB Sp	RPTH T(s)	RPTH Sp	IPTH T(s)	IPTH Sp
1	522,95	0,29	159,90	0,95	153,91	0,99	186,45	0,82
2	267,41	0,57	79,25	1,93	120,85	1,26	93,58	1,63
4	132,61	1,15	39,10	3,90	52,57	2,90	46,03	3,31
8	65,96	2,31	19,54	7,81	38,10	4,01	19,19	7,95
16	32,83	4,65	9,79	15,58	23,75	6,42	11,81	12,92

res. Al incrementar la dimensión del problema también aumenta el número de tareas independientes que pueden ser procesadas lo que incrementa el rendimiento paralelo (ver figura 4.2(b)).



(a) Resultados para la Instancia I3 (b) Resultados para la Instancia I4

Figura 4.2: Speedup para las instancias I3 e I4 en la máquina BullX.

4.4.2 Resultados para las instancias I3 e I4 en Aloe

Las tablas 4.8 y 4.9 y la figura 4.3 muestran la misma información que las tablas y figura que se han visto en la sección anterior pero para la máquina Aloe. Los tiempos de ejecución en Aloe son mejores, pero los speedup están más lejos del lineal que en BullX. Esto sucede porque la mejora de prestaciones en CPU de Aloe respecto a BullX es mayor que la mejora de prestaciones del subsistema de memoria. Esto provoca que los problemas limitados en memoria (memory bound), como el que se está tratando aquí, escalen peor en Aloe que en BullX. La tabla 4.7, muestra los DMIPS, calculados utilizando el *Dhrystone Benchmark*[1] [79], el ancho de banda de memoria (GB/s), calculado utilizando *pmbw*[2] y la relación entre el ancho de banda de memoria y los

Tabla 4.7: DMIPS y GB/s en BullX y Aloe además de la ganancia en DMIPS y GB/s de Aloe respecto a Bullx.

	Bullx	Aloe	Sp
DMIPS	28,59	51,92	1,81
GB/s	0,77	1,15	1,49
(GB/s)/DMIPS	0,027	0,022	0,81

[1] http://www.roylongbottom.org.uk/dhrystone results.htm
[2] https://panthema.net/2013/pmbw/

DMIPS, para BullX y Aloe. La columna Sp muestra la ganancia en DMIPS, GB/s y en la relación (GB/s)/DMIPS de Aloe sobre Bullx.

Tabla 4.8: Resultados en Aloe para la instancia I3. $t_{sec} = 11{,}07$ s.

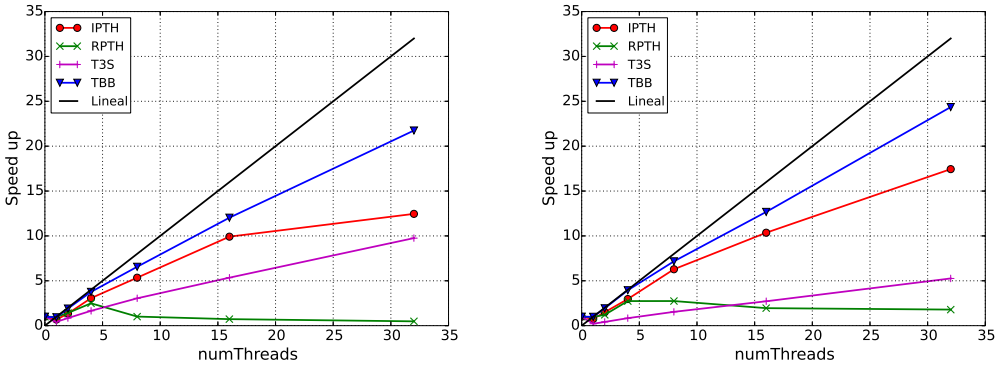
NTh	T3S T(s)	T3S Sp	TBB T(s)	TBB Sp	RPTH T(s)	RPTH Sp	IPTH T(s)	IPTH Sp
1	25,92	0,42	11,88	0,93	11,32	0,97	16,31	0,67
2	13,09	0,84	5,80	1,90	7,82	1,41	8,21	1,34
4	6,69	1,65	2,94	3,76	4,44	2,49	3,61	3,06
8	3,63	3,04	1,68	6,56	10,93	1,01	2,07	5,34
16	2,07	5,34	0,92	12,02	15,22	0,72	1,11	9,91
32	1,13	9,75	0,51	21,74	22,99	0,48	0,88	12,45

Tabla 4.9: Resultados en Aloe para la instancia I4. $t_{sec} = 57{,}75$ s.

NTh	T3S T(s)	T3S Sp	TBB T(s)	TBB Sp	RPTH T(s)	RPTH Sp	IPTH T(s)	IPTH Sp
1	269,84	0,21	58,54	0,98	59,66	0,96	77,44	0,74
2	138,68	0,41	29,87	1,93	47,71	1,21	38,50	1,49
4	69,05	0,83	14,63	3,94	21,08	2,73	19,46	2,96
8	37,47	1,54	8,06	7,16	21,05	2,74	9,18	6,28
16	21,25	2,71	4,56	12,67	29,59	1,95	5,57	10,35
32	10,98	5,26	2,37	24,35	32,22	1,79	3,31	17,44

4.4.3 Resultados para las instancias I5 e I6

Las instancias I5 e I6 se han utilizado para comprobar el comportamiento de los algoritmos cuando se encuentran a problemas de una dimensión mayor a 5. Un 6-símplex (7 dimensiones) tiene 21 lados y puede tener hasta 15 lados mayores. El nivel de paralelismo de las instancias crece con la dimensión, pero también lo hacen los requerimientos de memoria necesarios para resolver el problema. Para resolver estas instancias sólo nos basaremos en el algoritmo 4.4 (TBB) y en el algoritmo 4.8 (IPTH) ya que fueron las versiones paralelas más eficientes para resolver las instancias I3 e I4. Las tablas 4.10 y 4.11 muestran los resultados de la ejecución de los algoritmos TBB e IPTH en las



(a) Resultados para la Instancia I3

(b) Resultados para la Instancia I4

Figura 4.3: Speedup para las instancias I3 e I4 en la máquina Aloe.

Tabla 4.10: Resultados en Bullx para la instancia I5. $t_{sec} = 12.301{,}92$ s.

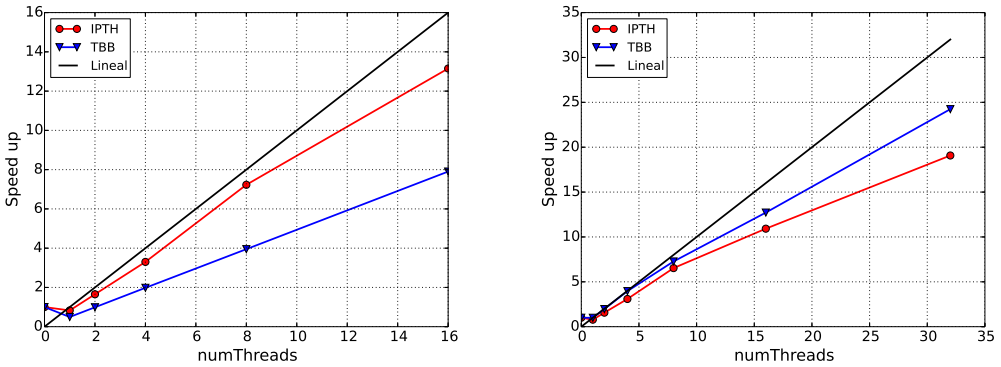
NTh	TBB T(s)	TBB Sp	IPTH T(s)	IPTH Sp
1	24.915,10	0,49	14.862,73	0,82
2	12.433,90	0,98	7.450,28	1,65
4	6.212,10	1,98	3.727,39	3,30
8	3.115,30	3,94	1.900,14	7,22
16	1.556,74	7,90	937,70	13,15

máquinas BullX y Aloe. IPTH es la mejor opción en BullX mientras que TBB es la mejor opción en Aloe. Esto sucede porque BullX sólo tiene 64GB de memoria RAM, mientras que Aloe tiene 256GB. TBB requiere más memoria que IPTH. TBB tiene una demanda de memoria que para estas instancias del problema hacen insuficientes los 64GB de RAM de BullX. Sin embargo los 256GB de Aloe son suficientes para el correcto funcionamiento de TBB, tal y como pasaba en las instancias I3 e I4.

Los algoritmos para resolver la instancia I6 tienen unos requerimientos de memoria muy altos. La versión de TBB termina antes de resolver el problema debido a que consume toda la memoria disponible del sistema. A pesar de la complejidad y los requerimientos de memoria tan altos de esta instancia, el modelo de búsqueda híbrido utilizado en el algoritmo IPTH tiene menos requerimientos de memoria que el usado en TBB y es capaz de resolver la instancia tanto en BullX como en Aloe. Los tiempos de ejecución y la ganancia de velocidad del algoritmo IPTH para resolver la instancia

Tabla 4.11: Resultados en Aloe para la instancia I5. $t_{sec} = 4.653{,}16$ s.

NTh	TBB T(s)	TBB Sp	IPTH T(s)	IPTH Sp
1	4.680,44	0,99	5.954,82	0,78
2	2.370,99	1,96	2.979,86	1,56
4	1.174,08	3,96	1.506,40	3,08
8	640,28	7,26	712,60	6,52
16	365,97	12,71	426,03	10,92
32	191,96	24,23	243,93	19,07



(a) Speedup para la instancia I5 en BullX.

(b) Speedup para la instancia I5 en Aloe.

Figura 4.4: Speedup para la instancia I5 en las máquinas Bullx y Aloe.

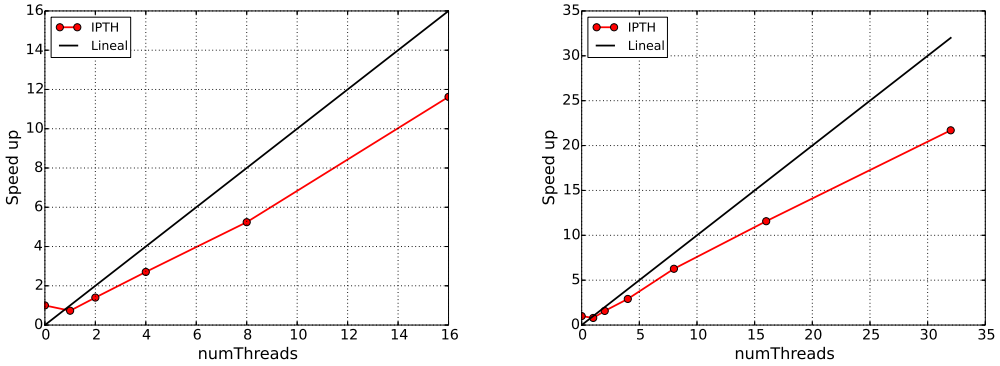
I6 se muestran en la tabla 4.13. La ganancia en velocidad aunque no es lineal, puede considerarse como buena.

4.5 Conclusiones

La respuesta a la pregunta 1.4, a partir de los experimentos realizados, es que los algoritmos TBB (algoritmo 4.5) e IPTH (algoritmo 4.8) son los que obtienen los resultados de una manera más eficiente. TBB es el algoritmo que mejor se comporta en problemas de dimensión menor que 5 debido a que no consume la memoria de las máquinas utilizadas. Sin embargo, debido a su alto requerimiento de memoria, para dimensiones mayores su comportamiento no es eficiente y es la versión IPTH la que obtiene los mejores resultados, aunque su buen speedup no es cercano al lineal.

Tabla 4.12: Resultados en BullX y Aloe para la instancia I6. $t_{sec-Bullx} = 15.566{,}24$ s. $t_{sec-Aloe} = 7.056{,}76$ s.

	BullX		Aloe	
NTh	IPTH T(s)	IPTH Sp	IPTH T(s)	IPTH Sp
1	21.328,00	0,72	8.918,79	0,79
2	11.116,86	1,40	4.484,77	1,57
4	5.753,37	2,70	2.425,28	2,90
8	2.970,66	5,23	1.125,51	6,26
16	1.339,23	11,62	609,90	11,57
32	-	-	325,20	21,69



(a) Speedup para la instancia I6 en BullX.

(b) Speedup para la instancia I6 en Aloe.

Figura 4.5: Speedup para la instancia I6 en las máquinas Bullx y Aloe.

Capítulo

5

Algoritmo para determinar el árbol con el menor número de clases de símplices

Para hacer una estimación del tamaño de un subárbol a partir de un nodo, el tener un árbol con el menor número de formas de símplices reduciría el número de posibles subárboles y por lo tanto, facilitaría el cálculo de una estimación de su tamaño. En este capítulo se va a dar respuesta a la pregunta 1.5: ¿cuál es el menor número de formas de los subsímplices en el refinamiento de un símplex regular mediante BLM? Para ello, se va a estudiar la forma que tienen los símplices resultantes del proceso de división. En la sección 5.1 se introducen los conceptos de clase de forma y medidas de calidad de un símplex. En la sección 5.2 se define como realizar la comparación de forma de símplices. Finalmente, en la sección 5.3 se presenta un método de obtención del árbol binario con el menor número de clases posibles y se da la secuencia de lados mayores a dividir que genera dicho árbol para un 3-símplex.

5.1 Clases de forma y medidas de calidad de un símplex

Generalmente, se requiere que los conjuntos de símplices obtenidos por el refinamiento sucesivo de un símplex inicial cumplan al menos estas dos condiciones: estabilidad y consistencia. La condición de estabilidad significa que dos símplices generados durante el proceso de refinamiento no deben degenerar, es decir, los ángulos interiores de todos los elementos deben estar delimitados uniformemente lejos del valor cero.

Trabajar con símplices que no sean degenerados es un pre-requisito a la hora de

lograr un buen resultado en cualquier análisis y/o procedimientos adaptativos que los utilicen como base [57]. Gracias a la propia definición de símplex sabemos que no van a aparecer símplices degenerados durante el proceso de bisección iterativa. Sin embargo, pueden aparecer símplices cuya forma, a pesar de no ser degenerada, se aproxime a ello. Se desean evitar a toda costa estos símplices ya que comprometen la condición de estabilidad anteriormente mencionada. Un símplex cuya forma tiende a la forma de un símplex degenerado tiene una forma "mala" o "pobre" . Durante el proceso de refinamiento de un símplex, la aparición de símplices de forma "mala" provoca que la forma de los símplices se deteriore con respecto a la forma del símplex inicial no refinado.

Por lo tanto, es muy importante contar con medidas de forma que nos informen si un símplex tiene una forma "buena" o "mala" especialmente durante el proceso de refinamiento. Podemos definir una medida de forma del siguiente modo:

Definición 2. *[12] Una medida de forma de un símplex es una función continua que evalúa la calidad de un símplex. Debe ser invariante respecto a traslación, rotación, reflexión y escalado uniforme del símplex. Debe tener valor máximo para el símplex regular y mínimo para un símplex degenerado. No debe haber otro máximo local diferente del máximo global para un símplex regular y no debe haber menor local alguno diferente del mínimo global del símplex degenerado. Para facilitar la comparación, se debe escalar la medida de calidad de modo que tenga valor 1 para el símplex regular y 0 para el símplex degenerado.*

Se pueden elegir muchos parámetros para medir la forma o la calidad de los símplices. Debido a esto, existen múltiples medidas de forma [63]. Por ejemplo, en [80] se utiliza como medida de calidad el grosor relativo de un símplex S definido por:

$$\rho(S) = r(S)/d(S) \tag{5.1}$$

donde r(S) es el radio de S interpretado como la distancia menor entre su centroide y cualquiera de las caras del símplex y d(S) es el diámetro de S, es decir, la longitud de su lado mayor. Stynes en [72], donde se analizó el método de bisección para triángulos, usó el cociente

$$t(S) = area(S)/d(S)^2 \tag{5.2}$$

Que se puede generalizar a símplices de una dimensión mayor n como

$$t(S) = volumen(S)/d(S)^n \tag{5.3}$$

En [5] se ha usado como medida de forma la relación entre la longitud del radio δ(S) de la esfera inscrita en el símplex y el radio R(S) de la esfera circunscrita en el símplex.

$$ratio(S) = \delta(S)/R(S) \tag{5.4}$$

Por lo tanto, existen muchas formas de medida de calidad de un símplex, si bien existen estudios que demuestran que la mayoría de estas medidas son equivalentes [13, 57, 63].

En esta tesis se ha desarrollado (ver [23]) un nuevo índice de calidad del símplex, llamado SQ, del inglés Simplex Quality, que se basa en el cálculo de la proporción entre las medias aritmética y geométrica del conjunto de ángulos que se forman entre los lados de un símplex. Sean $\alpha_1, \ldots, \alpha_m$ los ángulos entre los lados de un n-símplex. $SQ(S)$ se define como:

$$SQ(S) = m \cdot \frac{\sqrt[m]{\alpha_1^2 \cdot \alpha_2^2 \cdot \ldots \cdot \alpha_m^2}}{\alpha_1^2 + \alpha_2^2 + \ldots \alpha_m^2} \in (0, 1], \tag{5.5}$$

donde m es el número total de ángulos que viene determinado por

$$m = \frac{n(n+1)(n-1)}{2}. \tag{5.6}$$

Como la longitud de cada lado uniendo dos vértices es conocida, el ángulo α entre el par de lados (v_i, v_j) y (v_i, v_k) que comparten el vértice v_i puede ser determinado aplicando la ley de los cosenos. El valor del angulo α, tal como muestra la figura 5.1 se puede obtener mediante:

$$\alpha = \arccos\left(\frac{||v_i - v_j||^2 + ||v_i - v_k||^2 - ||v_j - v_k||^2}{2||v_i - v_j||||v_i - v_k||}\right). \tag{5.7}$$

$SQ(S) = 1$ si y solo si S es un símplex regular. Un símplex con una forma "mala" o "pobre" tiene ángulos muy pequeños en algunos de sus vértices y por tanto tendrá un valor de SQ muy próximo a cero. Se puede asegurar que dos símplices con distinto valor de SQ van a tener una forma diferente, pero tener el mismo valor de SQ no asegura que dos símplices tengan igual forma, es decir, tener igual valor de SQ es condición necesaria pero no suficiente para asegurar que dos símplices tienen igual forma.

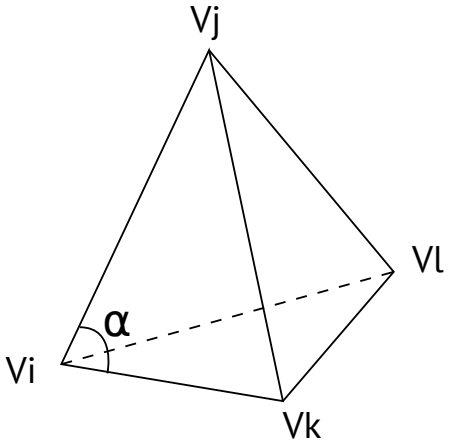


Figura 5.1: Obtención del ángulo α entre dos lados que comparten un vértice.

El estudio de las formas de los símplices nos va a ayudar a entender las características de los árboles binarios que aparecen durante el proceso de bisección de un símplex. Se desea conocer la forma de los símplices para poder desarrollar mecanismos de selección del lado mayor a dividir que logren que los símplices que aparecen durante el proceso no tengan una forma "mala" o "pobre" . Además se desea que el árbol binario generado durante el proceso de bisección contenga el menor número de formas de símplices posible.

Definición 3. *Dos símplices A y B tienen la* misma forma *si es posible conseguir que cada uno de los vértices de A coincida con un vértice de B después de un proceso de escalado, traslación y una operación de matrices ortogonales que incluye rotación y reflejo, ver [63].*

Comprobar que dos símplices tienen la misma forma no es tarea sencilla ya que el número de posibles coincidencias es $(n+1)!$.

Observación 5.1. *El coste computacional del cálculo y comparación de SQ para dos símplices es mucho menor que averiguar si la forma de ambos símplices es la misma. (ver definición 3).*

Por tanto, si se desea averiguar si dos símplices son semejantes en forma, es más sencillo comprobar primero si poseen igual valor de SQ, y en caso afirmativo, proceder con la comparación de forma.

Una cuestión sin resolver es, si dentro de un mismo árbol, es posible encontrar símplices con forma diferente y mismo valor de SQ. Se sabe que es posible generarlos de manera individual, pero no se ha podido comprobar su existencia dentro de un árbol binario generado a partir de un símplex regular. Si fuese posible que, dentro del árbol binario creado por refinamiento de un símplex regular por bisección del lado mayor, dos símplices cualesquiera con igual SQ tuvieran siempre igual forma, se reduciría drásticamente el tiempo de computación para comprobar la semejanza de forma.

5.2 Medida de semejanza de forma entre dos símplices

Existen varios modos de comprobar si dos símplices poseen la misma clase de forma de acuerdo con la Definición 3. Específicamente, se va a comprobar la definición para n-símplices considerando su análisis no solo en un espacio $n+1$-dimensional sino también en uno n-dimensional.

Consideremos dos n-símplices S e Y con sus correspondientes conjuntos de vértices V y W, o lo que es lo mismo, $S = conv(V)$ e $Y = conv(W)$. En un contexto algorítmico, V y W son almacenados como $(n+1)$ matrices ordenadas. La definición de semejanza presentada previamente puede ser reescrita de un modo más exacto utilizando tan solo transformaciones afines de matrices (escalado, traslación, rotación y reflejo) para poder trabajar en este contexto.

Definición 4. *Dos símplices $S, Y \subset \mathbb{R}^{n+1}$ tiene la misma forma si existe un factor de escala $a > 0$, un vector de traslación $b \in \mathbb{R}^{n+1}$ y una matriz ortonormal de rotación/-reflejo R, tales que $\forall w \in Y\ \exists v \in S$ con $w = b + aRv$.*

Para el caso práctico, escalado y traslación pueden ser realizados a través de la normalización de los símplices S e Y. Tras centrar los símplices en el origen de coordenadas, ambos se pueden escalar para que su lado mayor tenga longitud igual a 1. Para ello basta dividir todos los lados de un símplex por el tamaño de este. Los lados del símplex S serán divididos por $\omega(S)$ y los lados de Y por $\omega(Y)$. Si se consideran V y W como matrices, el proceso de escalado y traslación produce las siguientes matrices:

$$\hat{V} = \frac{1}{\omega(S)}(V - \frac{1}{n+1}V\mathbf{1}\mathbf{1}^T),\ \hat{W} = \frac{1}{\omega(Y)}(W - \frac{1}{n+1}W\mathbf{1}\mathbf{1}^T), \tag{5.8}$$

donde $\mathbf{1}$ es el vector unidad. A continuación se muestra que si los símplices $\hat{S} = conv(\hat{V})$ y $\hat{Y} = conv(\hat{W})$ son iguales, entonces S e Y también son semejantes.

Proposición 5.1. *Sean V y W matrices $(n+1)\times(n+1)$ no singulares y $\hat{V}$ y $\hat{W}$ matrices que cumplen la ecuación (5.8). Si existe una matriz ortonormal R tal que $\hat{W}$=$R\hat{V}$ entonces los símplices S=$conv(V)$ e Y=$conv(W)$ tienen la misma forma.*

Demostración Consideremos $a = \frac{\omega(Y)}{\omega(S)}$ y $b = \frac{1}{n+1}W\mathbf{1} - \frac{a}{n+1}RV\mathbf{1}$. Se debe probar que $\forall w \in Y\ \exists v \in S$ tal que $w = b + aRv$. Cualquier $w \in Y$ se puede definir como la combinación convexa de sus puntos extremos: $w = W\lambda$ con $\sum_i \lambda_i = 1$ y $\lambda_i \geq 0,\ i = 1,\ldots,n+1$. Ahora consideremos $v = V\lambda$, $\hat{v} = \hat{V}\lambda$ y $\hat{w} = \hat{W}\lambda$. Entonces se deduce de la condición y $\sum_i \lambda_i = 1$ que

$$\hat{w} = R\hat{v} \Rightarrow\ \omega(Y)\hat{W}\lambda = \omega(Y)R\hat{V}\lambda = \tfrac{\omega(Y)}{\omega(S)}R(V\lambda - \tfrac{1}{n+1}V\mathbf{1}) \Rightarrow$$
$$w = W\lambda = \omega(Y)\hat{W}\lambda + \tfrac{1}{n+1}W\mathbf{1} = \tfrac{\omega(Y)}{\omega(S)}Rv - \tfrac{\omega(Y)}{\omega(S)}R\tfrac{1}{n+1}V\mathbf{1} + \tfrac{1}{n+1}W\mathbf{1} = aRv + b.$$

□

Como en nuestro razonamiento se combinan ideas sobre conjuntos de vértices y representación de matrices, incluso si $\hat{W} = R\hat{V}$, aun necesitaríamos encontrar el orden correcto de los vértices (permutación) para poder comprobar que nuestro razonamiento es correcto. Otro reto es que la matriz R rota y refleja alrededor del eje $\mathbb{O} = \{x \in \mathbb{R}^{n+1} | x = \alpha\mathbf{1}, \alpha \in \mathbb{R}\}$ que atraviesa el origen. La imagen de R con respecto a la matriz V esta por tanto en el plano dimensional inferior $\mathbb{B} = \{x \in \mathbb{R}^{n+1} | \sum_i x_i = 0\}$. Por lo tanto, $\mathbf{1}$ debe ser un autovector con autovalor 1 de R; $R\mathbf{1} = \mathbf{1}$. Las matrices $\hat{V}$ y $\hat{W}$ son singulares ya que sus columnas suman 0. Aun así, para cualquier escalar α distinto de cero, $\hat{W} + \alpha\mathbf{1}\mathbf{1}^T = R(\hat{V} + \alpha\mathbf{1}\mathbf{1}^T)$, de modo que R se puede calcular mediante

$$R_{n+1} = (\hat{W} + \alpha\mathbf{1}\mathbf{1}^T)(\hat{V} + \alpha\mathbf{1}\mathbf{1}^T)^{-1}. \tag{5.9}$$

Incluso si $\hat{S} = conv(\hat{V})$ e $\hat{Y} = conv(\hat{W})$ tienen la misma forma, aun se debe buscar la correspondencia correcta de los vértices $\{v_1, v_2, \ldots, v_{n+1}\}$ para poder comprobar dicha igualdad. Para cualquier permutación se puede calcular la matriz R_{n+1}; si esta es ortonormal para una de las permutaciones, entonces los símplices S e Y tienen la misma forma.

En lugar de trabajar en un espacio $n+1$ dimensional, se puede encontrar también la rotación-reflexión en un espacio n-dimensional. Sea $\mathbb{B} = \{x \in \mathbb{R}^{n+1} | \sum_i x_i = 0\}$

el espacio lineal donde se pueden encontrar $\hat{S} = conv(\hat{V})$ y $\hat{Y} = conv(\hat{W})$ y sea B una matriz que contiene una base de $\mathbb{B}$. Esto significa que el símplex $conv(\hat{W})$ esta en el espacio determinado por las columnas de B; $conv(\hat{W}) \subset < B >$. Considerando una base ortonormal, se puede realizar el análisis en un espacio n-dimensional.

Proposición 5.2. *Sea B una base ortonormal de $\mathbb{B}$, sean V y W dos matrices no singulares $(n+1) \times (n+1)$ y $\hat{V}$ y $\hat{W}$ las descritas en la ecuación (5.8). Si existe una matriz $n \times n$ de rotación/reflexión R ($det(R) = \pm 1$ y $RR^T = I$) tal que $B^T\hat{W} = RB^T\hat{V}$, entonces los símplices $conv(V)$ y $conv(W)$ tienen la misma forma.*

Demostración Sean Z y X las soluciones (matrices $n \times (n+1)$) de $\hat{W} = BZ$ y $\hat{V} = BX$. Las columnas son afinmente independientes y suman cero; $Z\mathbf{1} = X\mathbf{1} = \mathbf{0}$. Existe una matriz $n \times n$ de rotación/reflexión R tal que $Z = RX$. De acuerdo con la definición, $conv(Z)$ y $conv(X)$ tienen la misma forma en $\mathbb{R}^n$. Al multiplicar por la base B se obtiene $BZ = \hat{W} = BRX = BRB^T\hat{V}$. La matriz BRB^T no es de rango completo; el autovalor en la dirección $\mathbf{1}$ es cero. Haciendo las matrices de rango completo mediante $\hat{B} = \left(B\,, \frac{1}{\sqrt{n+1}}\mathbf{1}\right)$ y $\hat{R} = \begin{pmatrix} R & \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix} \\ 0, \ldots, 0 & \end{pmatrix}$ se proporciona $\hat{W} = BRX = \hat{B}\hat{R}\hat{B}^T\hat{V}$. El determinante de $\hat{R}$ es $1 \times \det(R)$. Además, $\hat{R}, \hat{B}$ son ortonormales y de rango completo: $\hat{B}^{-1} = \hat{B}^T$. Tenemos que $\hat{B}\hat{R}\hat{B}^T(\hat{B}\hat{R}\hat{B}^T)^T = \hat{B}\hat{R}\hat{B}^T(\hat{B}\hat{R}^T\hat{B}^T) = I$. Por lo tanto, ya que $\hat{B}\hat{R}\hat{B}^T$ es una matriz ortonormal de rotación/reflexión, acorde con la Proposición 5.1, $conv(V)$ y $conv(W)$ tienen la misma forma. □

A continuación se muestra un modo de construir una base ortogonal. Para $n = 2$ se puede escribir dicha base como:

$$B_2 = \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix} \right)$$

y recursivamente definir

$$B_n = \begin{pmatrix} B_{n-1} & \frac{1}{\sqrt{n(n+1)}} \begin{pmatrix} \mathbf{1} \\ -n \end{pmatrix} \\ 0, \ldots, 0 & \end{pmatrix}.$$

Por ejemplo,

$$B_3 = \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ -2 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{12}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ -3 \end{pmatrix} \right)$$

$$B_4 = \left(\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ -2 \\ 0 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{12}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ -3 \\ 0 \end{pmatrix}, \frac{1}{\sqrt{20}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -4 \end{pmatrix} \right)$$

La proposición 5.2 muestra que también se puede usar el algoritmo RS en un espacio dimensional inferior ahorrando de esta forma memoria de almacenamiento y operaciones computacionales. Trabajando con $n+1$ vértices en un espacio n-dimensional, para recuperar R, uno de los vértices se puede obviar y se puede resolver la igualdad de la misma forma que en (5.9). Alternativamente, uno puede usar un enfoque de mínimos cuadrados siguiendo la proposición 5.3.

Proposición 5.3. *Sean Z y X dos matrices $n \times (n+1)$ de rango n. Si existe una matriz $n \times n$ de rotación/reflexión R (det$(R) = \pm 1$ y $RR^T = I$) tal que $Z = RX$, entonces $R = ZX^T(XX^T)^{-1}$.*

Demostración Si $Z = RX$ entonces $ZX^T(XX^T)^{-1} = RXX^T(XX^T)^{-1} = R.$ □

Retomando los datos en un espacio $n+1$ dimensional, si en la proposición 5.3 se consideran las matrices $Z = B_n^T \hat{W}$ y $X = B_n^T \hat{V}$ entonces se puede obtener la matriz de rotación/reflexión mediante:

$$R_n = B_n^T \hat{W} \hat{V}^T B_n (B_n^T \hat{V} \hat{V}^T B_n)^{-1}. \tag{5.10}$$

En lugar de centrase en la representación matricial del conjunto de vértices, uno puede observar los n lados de un símplex en un espacio n-dimensional conectados al primero (o a cualquier) vértice.

Proposición 5.4. *Sean Z y X dos matrices $n \times (n+1)$ cuyas columnas son conjuntos afinmente independientes y sean F y E matrices de lados $n \times n$ definidas por las columnas $f_i = z_{i+1} - z_1, i = 1, \ldots, n$ y $e_i = x_{i+1} - x_1, i = 1, \ldots, n$. Si existe una matriz $n \times n$ no singular R tal que $Z = RX$, entonces $F = RE$ y $R = FE^{-1}$.*

Demostración Sigue de escribir $f_i = z_{i+1} - z_1 = R(x_{i+1} - x_1) = Re_i.$ □

En la práctica la proposición 5.4 puede ser usada eligiendo dos vértices que coincidan en $B^T\hat{W}$ y $B^T\hat{V}$ y mediante la construcción del conjunto de lados F y E. El cálculo

$$R_n = FE^{-1} \tag{5.11}$$

requiere menos operaciones computacionales que (5.10). La principal preocupación es encontrar el orden correcto, es decir, la permutación de vértices. Teniendo esto en cuenta, se debe encontrar la matriz de permutación P correcta, calculando o R_{n+1} en (5.9) o R_n en (5.10) o (5.11) y comprobando si es ortonormal. Como hay $(n+1)!$ matrices de permutación P diferentes, R deberá ser recalculada y comprobada $(n+1)!$ veces en el peor caso.

5.3 Algoritmo para determinar el árbol con el menor número de clases

La cuestión que se va a abordar en este apartado es cómo se debe generar el árbol binario mediante el algoritmo 1.2 para que aparezcan el menor número de clases de símplices posible. Para dar respuesta a esta cuestión se ha desarrollado un algoritmo especifico de B&B llamado (MinClassB&B) que se muestra en el algoritmo 5.1. En dicho algoritmo, cada nodo se corresponde con un árbol T cuya raíz es el símplex regular de lado unidad. Cada nodo tiene siguiente información:

$\Lambda(T)$: Conjunto de hojas del árbol T, son aquellos nodos que están pendientes de ser procesados.
$C(T)$: Es el conjunto de clases que ya se han encontrado en los nodos del árbol T.
$c(S)$: Es la forma del símplex S.

El algoritmo MinClassB&B básicamente construye un árbol de árboles (ToT) con un grado de ramificación que varía en cada nodo. Se puede observar un ejemplo en la figura 5.2. El conjunto de hojas de ToT es almacenado en Γ. Se aplicarán los siguientes conceptos referentes a los algoritmos de Ramificación y Acotación.

- MinNC es el límite superior del número mínimo de clases. Inicialmente, MinNC puede inicializarse a infinito o con un valor encontrado en cálculos anteriores y

que se desee mejorar. Por ejemplo, se sabe que que el número máximo de clases para un 3-símplex es 8 [23].

- Si para un árbol T, el número de clases $|C(T)|$ es mayor que MinNC, el árbol T puede descartarse de la búsqueda porque, aunque se continúe con su ramificación, su número de clases de símplices va a ser mayor que la de otro árbol que actualizó MinNC.
- Cuando el refinamiento de T no produce nuevas clases de símplices, el árbol se puede eliminar. Este concepto se muestra en la figura 5.2.

La idea general es que solo se deben almacenar símplices como hojas en $\Lambda(T)$, si su forma no ha sido encontrada previamente.

El conjunto Γ inicialmente tiene solo un árbol T que esta formado por el conjunto Λ con una hoja que es el símplex inicial. El resto del algoritmo sigue la estructura general de cualquier algoritmo de Ramificación y Acotación. No se han proporcionado detalles de qué ramificación es la que se realiza primero o qué árbol se selecciona primero. El nuevo conjunto de formas $C(X)$ se define en la línea 18 donde $|C(X)|$ es una cota inferior del número de formas del árbol. Para reducir la complejidad del pseudocódigo, queda implícito que una hoja de ToT se almacena primero en Γ antes de que pueda saberse si es posible rechazarla porque su mínimo número de clases ya supera el límite superior (línea 6), o T ya no tiene más hojas (línea 7). El límite superior del número mínimo de clases de símplices solo se puede actualizar con el número de formas de un árbol T sin hojas en la línea 8. Seleccionar un límite superior inicial en lugar de establecer su valor a infinito reduce drásticamente el coste computacional del algoritmo.

La parte computacionalmente mas compleja del algoritmo 5.1 es la comparación de las formas de los nuevos símplices con las clases ya existentes en $C(T)$. Además, (en la línea14) el algoritmo comprueba si dos símplices hijos tienen la misma forma para, en este caso, eliminar un hijo y analizar tan solo el otro, como se muestra en el árbol de la figura 5.2. Adicionalmente, un árbol no va a ser analizado si su hermano produce la misma combinación de nuevos símplices. Este concepto no está expresamente descrito en el algoritmo 5.1. La figura 5.2 muestra un ejemplo de ello.

El algoritmo MinClassB&B ha sido ejecutado para un 2-símplex y un 3-símplex regulares de lado unidad. El 4-símplex es un caso computacionalmente complicado de-

Algoritmo 5.1 MinClassB&B(S_1)

Requiere: S_1: el símplex inicial.

$\Lambda(T_1)$ consiste en una sola hoja, S_1 y $C(T_1) = \{c(S_1)\}$
$\Gamma := \{T_1\}$ ▶ Conjunto de árboles
MinNC:=∞ ▶ Número Minimo de Clases ya encontradas
while $\Gamma \neq \emptyset$ **do**
  Extraer un árbol T de Γ
  if $|C(T)| \leq$ MinNC **then**
    if $\Lambda(T) = \emptyset$ **then**
      MinNC:=$|C(T)|$
      Almacenar la regla de selección $SelectLE$ aplicada para generar T
    else
      Extraer un símplex S_i de $\Lambda(T)$
      for todos los lados mayores (j, k) de S_i **do**
        Generar nuevas hojas $\{\sigma_1, \sigma_2\}$:=Bisect(S_i, j, k)
        if $c(\sigma_1) = c(\sigma_2)$ **then**
          Eliminar una de las hojas, por ejemplo σ_2
        Crear un nuevo árbol X
        Conjunto de hojas $\Lambda(X) := \{\sigma_l,\ c(\sigma_l) \notin C(T)\} \cup \Lambda(T)$,
        y conjunto de formas $C(X) := \{C(T)\} \cup \{c(\sigma_l) \notin C(T)\}$
        Almacenar X en Γ
        Mantener la traza de los lados seleccionados $SelectLE(S_i)$
return $MinNC$ y la regla de selección $SelectLE$ para alcanzarlo

bido a la gran cantidad de posibilidades de selección de los diferentes lados mayores. Debido a esto, va a ser necesario el uso de computación de altas prestaciones para encontrar una solución en un tiempo de cómputo razonable, así como para poder abordar problemas de mayor dimensión.

5.4 Resultados

El resultado para un 2-símplex muestra que en el árbol con el menor número de clases existen tres formas diferentes de símplices, tal y como se puede observar en la figura 1.7. Mediante el uso de SelectLE(S_i)= $\{1, 2\}$ en el proceso RS, se evita el cálculo

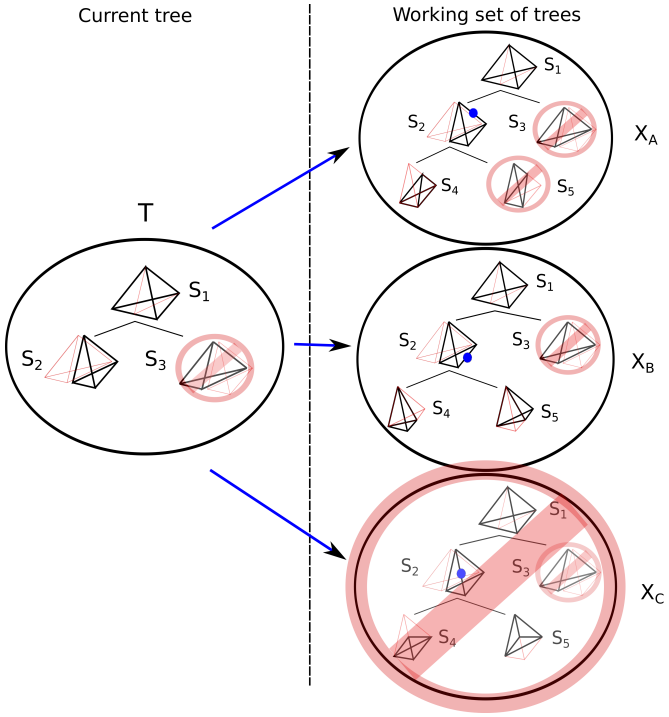


Figura 5.2: Generación de nuevas hojas en el árbol de árboles (ToT) cuando existen varios lados mayores de S_2. X_C no se va a analizar porque X_B tiene las mismas formas en los símplices descendientes que S_2.

de longitudes de los lados.

Para el 3-símplex regular se ha encontrado que el número mínimo de clases que se pueden obtener es ocho. Sólo existe un árbol binario como solución del algoritmo y por lo tanto, una sola regla de selección del lado mayor que permite obtener el árbol binario con menor número de clases. Se puede comprobar que los métodos de selección del lado mayor LEB$_\alpha$ ([23]), LEB$_C$ y LEB$_M$ ([24]) son óptimos para un 3-símplex.

La tabla 5.1 muestra los valores usados en SelectLE() para la obtención del árbol binario con menor número de clases. Desde el nivel 5 del árbol binario, el lado mayor que debe ser seleccionado en cada símplex se repite con una periodicidad de tres niveles. Por lo tanto, los símplices del nivel 8 deberán ser divididos por el mismo lado que los del nivel 5, los del nivel 9 igual que los del 6 y los del nivel 10 del mismo modo que los del nivel 7. Es importante resaltar que i es el índice del símplex S_i que está en el nivel L=$\lceil \log_2\ i \rceil$ del árbol binario que se ha generado al aplicar el algoritmo RS y

que los vértices de S_i son reordenados siguiendo el algoritmo 1.3.

Tabla 5.1: Secuencia de LMs a usar en SelectLE(S_i) del algoritmo 1.2, que ha sido obtenida mediante el algoritmo 5.1 para obtener el árbol con el menor número de clases en un 3-símplex.

Nivel	SelectLE(S_i)
1	$\{1, 2\}$
2	$\{2, 3\}$
3	$\{1, 2\}$
4	$\{1, 2\}$, para i pares $\{1, 3\}$, para i impares
5	$\{1, 2\}$
6	$\{1, 2\}$
7	$\{1, 3\}$, para i pares $\{1, 2\}$, para i impares
8	Como el nivel 5
9	Como el nivel 6
10	Como el nivel 7 ...

La figura 5.3 muestra el árbol binario resultante donde los símplices cuya forma ya haya sido encontrada no son divididos nunca más para ilustrar la regla optima de selección del lado mayor. Para un 3-símplex se puede llegar a este resultado aplicando las reglas LEB_α, LEB_C y LEB_M, pero cuando aumentamos las dimensiones del símplex, por ejemplo usando un 4-símplex, estas reglas ya no son válidas pues a pesar de su aplicación existen varios lados mayores detectados como óptimos y por tanto sigue existiendo ambigüedad en el proceso de selección del lado mayor.

5.5 Conclusiones

En este capítulo, se ha dado respuesta a la pregunta 1.5 para un 3-símplex ya que se ha obtenido el árbol con el menor número de 8 clases de símplices [23]. Se ha calculado en este caso una secuencia de lados mayores que llevan a la obtención de dicho árbol

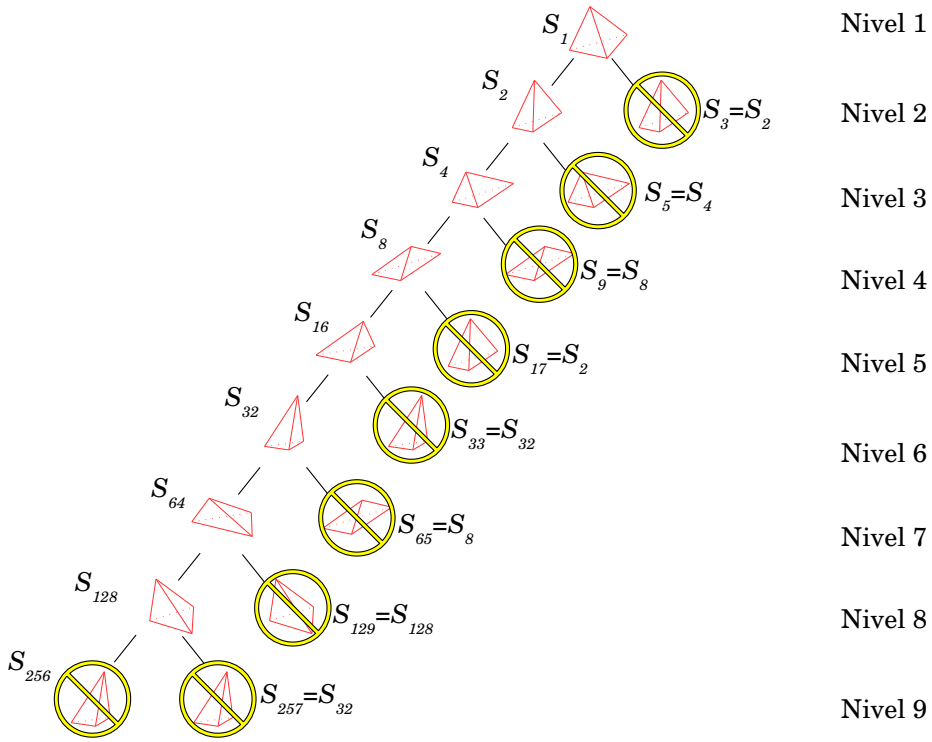


Figura 5.3: Árbol binario generado por algoritmo RS (véase el algoritmo 1.2) para un 3-símplex usando en SelectLE() la tabla 5.1.

sin necesidad de realizar cálculos para la obtención del lado mayor. Para dimensiones mayores a cuatro, debido al alto coste computacional del algoritmo, se requiere el uso de computación de altas prestaciones para poder obtener una solución en un tiempo razonable. Este estudio se abordará en trabajos futuros.

Capítulo

6

Algoritmo para encontrar la menor secuencia de lados mayores que genera el menor árbol

En el refinamiento de un símplex mediante BLM, hay que realizar repetidamente el cálculo de distancias para conocer los lados mayores susceptibles de ser divididos. Para reducir el coste computacional de este tipo de algoritmos se puede proporcionar una secuencia de lados mayores a dividir que genere el menor árbol binario. Estamos interesados en encontrar la secuencia recurrente de menor tamaño. En este capítulo se da respuesta a la cuestión 1.6: ¿cuál es la secuencia recurrente de lados mayores a dividir y de menor tamaño para obtener el árbol binario de menor tamaño?.

6.1 Condición m_k-válido

Para dimensiones superiores a 3, durante el proceso de bisección iterativa de un símplex regular, se pueden generar diferentes árboles binarios con el menor número de símplices. A cada uno de estos árboles se le va a llamar Menor Árbol Binario (MAB). Dado un MAB, se puede determinar la Secuencia de Lados Mayores, S-LM, como un vector de lados mayores a dividir que será usado en el algoritmo 1.2, en la regla de división, línea 6. El elemento S-LM$_i$ especifica uno de los lados mayores de S_i, el lado mayor por el que deberá dividirse el símplex para aplicar la secuencia. El número de elementos del vector S-LM es el mismo número que el número de nodos (sin incluir los nodos

hojas) del correspondiente MAB. Este número se incrementa conforme la dimensión del problema crece y el valor de ϵ se decrementa. Por lo tanto, este número puede ser muy grande.

Dada la numeración de símplices del algoritmo 1.2, se puede determinar el nivel al que pertenece un símplex S_i dentro del árbol binario con la formula $\ell = \lceil 1 + \log_2 i \rceil$. Nos centramos en la idea de que el índice del lado mayor a dividir será el mismo en varios símplices del mismo nivel de un árbol binario, debido a la reordenación del índice de los vértices presentada en el algoritmo 1.3. Por lo tanto, se establecen algunas condiciones que deben cumplir los índices de los lados mayores a dividir en los símplices de un mismo nivel del árbol binario.

Definición 6.1. *Dos símplices tienen un índice de lado mayor h en común cuando el lado $E_h = \{j, k\} = \{k, j\}$ es un lado mayor en ambos símplices. El índice de un lado mayor puede pertenecer a lados no compartidos entre los símplices.*

Definición 6.2. *Sea $m \in \mathbb{N}^+$. El símplex i del nivel ℓ está en el subconjunto M_j, $j = 0, \ldots, m-1$, si $j = i \bmod m$.*

Definición 6.3. *El subconjunto M_j del nivel ℓ (ver definición 6.2) se denomina válido si en él existe un índice de lado h tal que E_h es un lado mayor para todos los símplices en M_j.*

M_j puede ser válido por distintos índices de LM.

Definición 6.4. *Un nivel ℓ en un árbol binario se denomina m-válido si todos los subconjuntos $M_j,\ j = 0, \ldots, m-1$ del nivel ℓ son válidos (ver definición 6.3).*

Un índice de lado h que valide el subconjunto M_j puede ser diferente del índice de lado g que válida el subconjunto $M_k,\ j \neq k$. Un nivel m-válido puede contener menos de m símplices.

Definición 6.5. *Un árbol binario con $L+1$ niveles será m-válido cuando todos sus primeros L niveles sean m-válidos (ver definición 6.4).*

El conjunto de índices de LMs que validen un nivel del árbol binario puede ser diferente del conjunto de índices de LM que validen otro nivel del árbol binario (AB).

Definición 6.6. *Dado un MAB m-válido con $L+1$ niveles, se define la matriz A de tamaño $L \times m$ con los elementos $a_{\ell,j}$ que denotan un índice h de un LM que valide el subconjunto M_j del nivel ℓ.*

No todos los elementos de la matriz A tienen por que tener un valor, debido a la ausencia de los correspondientes nodos en el árbol binario.

Observación 6.1. *Un árbol binario generado por BLM, con $L+1$ niveles y un máximo de 2^{L-1} elementos en el nivel L, siempre será m-válido para $m = 2^{L-1}$.*

Observación 6.2. *Dado $k \in \mathbb{N}$, un árbol binario 2^k-válido es también 2^{k+1}-válido.*

La Propiedad 6.2 y el número de nodos $2^{\ell-1}$ en el nivel ℓ de un árbol binario sugieren usar $m = 2^k$. Un vector S-LM tiene unos requerimientos de memoria menores que una matriz $\mathrm{A}_{(L\times m)}$ con $m = 2^{L-1}$. Sin embargo, estamos interesados en valores de $k < L-1$.

Definición 6.7. *Un árbol binario es m_k-válido si es m-válido (ver definición 6.5) con $m = 2^k$, $k \in \mathbb{N},\ k < L-1$.*

Por ejemplo, en la tabla 6.1 (página 94), los niveles 1 a 3 son 1-válidos y el nivel 4 es 2-válido.

Definición 6.8. *Un árbol binario que no es m_k-válido se denomina m_k-inválido.*

Observación 6.3. *Un árbol binario generado a partir de un árbol binario m_k-inválido que incrementa su número de niveles, es m_k-inválido.*

La existencia de un MAB m_k-válido depende de las combinaciones de índices de lado mayor que que hagan el nivel del MAB m_k-válido, esto también depende de las combinaciones seleccionadas en los niveles previos. En el peor de los casos, el número de combinaciones de índices de lado mayor que hacen que un nivel sea m_k-válido viene determinado por la suma de las posibles combinaciones que forman el nivel 1-válido, 2-válido, ... y 2^k-válido:

$$\sum_{i=0}^{k} \binom{nl}{2^k} \tag{6.1}$$

siendo $n = \frac{n(n+1)}{2}$ el número de lados de un n-símplex Para encontrar todos los MABs m_k-válidos, necesitamos un algoritmo que genere todos los posibles MABs y un algoritmo para desechar aquellos árboles que sean m_k-inválidos. Las siguientes secciones describen estos algoritmos.

6.2 Algoritmo para generar todos los menores árboles binarios (TMAB)

El algoritmo TMAB almacena todos los MABs en un solo árbol general (GT, del ingles General Tree). De esta forma, solamente los lados mayores que alcancen MABs permanecerán en GT. Cada nodo N de GT contiene la siguiente información:

- $S = conv(V)$: El símplex actual.
- El conjunto de lados mayores de S, con la siguiente información para cada lado mayor $E_h = \{j, k\}$:
  - R_h: El tamaño total de los dos subárboles generados.
  - N_{lh}: Nodo izquierdo.
  - N_{rh}: Nodo derecho.

Algoritmo 6.1 TMAB (S, ϵ)

Requiere: S: símplex, ϵ: precisión

1: **if** $w(S) \leq \epsilon$ **then**
2: **return** 1
3: Crear un nuevo nodo N con S
4: **for** cada lado mayor $E_h = \{j, k\}$ de S **do**
5: $\{S_l, S_r\}$:= Bisección(S, j, k)
6: $\{r_l, N_{lh}\}$:= TMAB(S_l, ϵ)
7: $\{r_r, N_{rh}\}$:= TMAB(S_r, ϵ)
8: $R_h := r_l + r_r$
9: Añadir $(E_h, N_{lh}, N_{rh}, R_h)$ a N
10: TamMinimo := $\min_h\{R_h\}$
11: **for** cada lado mayor E_h **do**
12: **if** $R_h >$ TamMinimo **then**
13: Quitar subárboles almacenados en N_{lh} y N_{rh}
14: Quitar $(E_h, N_{lh}, N_{rh}, R_h)$ de N
15: **return** 1 + TamMinimo, N ▶ Vuelve a las líneas 6 o 7

El pseudocódigo del algoritmo TMAB se muestra en el algoritmo 6.1. Este algoritmo es una extensión del algoritmo 3.1, donde solo aquellos LM que generan un MAB

son almacenados en el GT. Los parámetros de entrada son el símplex regular inicial S_1 y la precisión deseada ϵ. La figura 6.1 muestra los primeros niveles de GT para un 3-símplex.

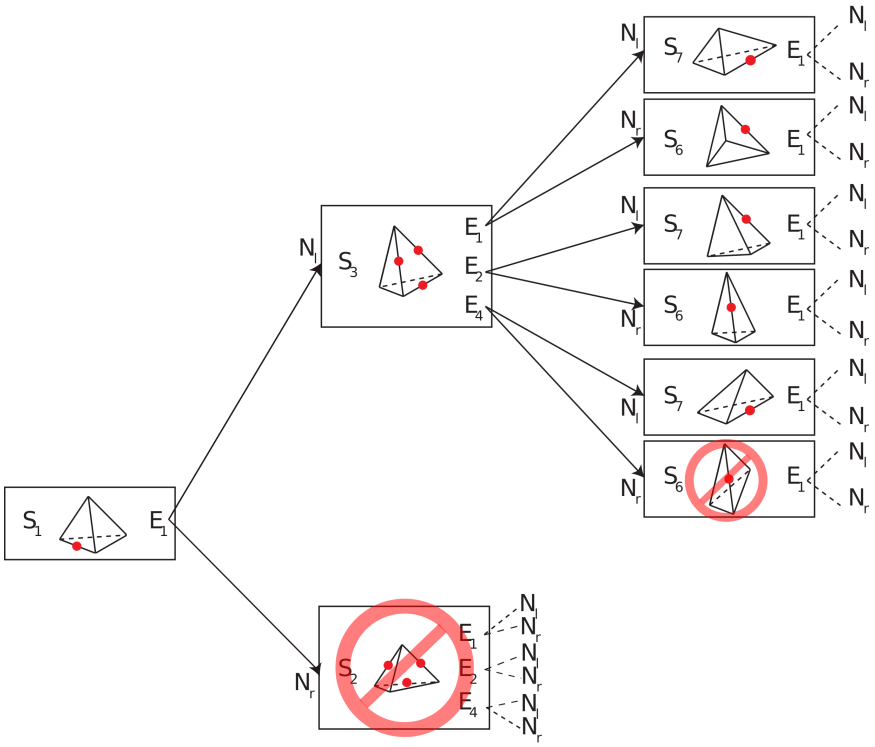


Figura 6.1: GT para un 3-símplex. Solo se evalúa un lado de S_1 porque es un símplex regular. S_2 se genera pero no se evalúa porque es simétrico a S_3. Lo mismo ocurre con los símplices S_6 y S_7.

Los resultados del algoritmo 6.1 son el tamaño de un MAB generado por BLM y el nodo raíz del árbol GT, N_1 contiene S_1. Siguiendo la numeración de los símplices expuesta en el algoritmo 1.2, donde dividiendo un símplex S_i por BLM se generan dos subsímplices S_{2i} y S_{2i+1}, pueden existir muchas instancias de S_i en GT. Cada instancia depende de el lado mayor biseccionado en niveles superiores del árbol GT. En el algoritmo 6.1, línea 12, aquellos lados mayores y subárboles generados a partir de ellos, que no dan como resultado un MAB, son eliminados del GT.

6.3 Algoritmo para contar el número de menores árboles binarios en el GT (NMAB-GT)

El algoritmo 6.2 cuenta el número de MABs en GT. No cuenta todos los posibles MABs porque se eliminan las simetrías (ver figura 6.1). El algoritmo cuenta los árboles binarios de forma recursiva desde las hojas hacia nodo raíz. El número de MABs en cada nodo de GT depende del número de arboles binarios en cada uno de sus descendientes, N_{lh} y N_{lh} que se formaron dividiendo en el símplex de N por el lado mayor E_h, y la combinación de los subárboles. Por ejemplo, si tenemos un nodo con dos hijos, uno que genera dos subárboles binarios (tiene al menos dos lados mayores) y otro que genera tres subárboles binarios (tiene al menos tres lados mayores), el número de subárboles binarios generados en el nodo actual es seis. La combinación de los subárboles en el algoritmo NMAB-GT comienza con el nodo raíz del árbol GT, resultado del algoritmo 6.1.

Algoritmo 6.2 NMAB-GT (N)

Requiere: N: un nodo del árbol GT

1: **if** N es una hoja **then**
2: **return** 1
3: $R := 0$ ▶ Número de Menores Subárboles Binarios (MSABs) bajo el nodo N
4: **for** cada lado mayor E_h de S en el nodo N **do**
5: r_l := NMAB-GT(N_{lh}) ▶ Número de MSABs en el subárbol izquierdo
6: r_r := NMAB-GT(N_{rh}) ▶ Número de MSABs en el subárbol derecho
7: $R := R + r_l * r_r$
8: **return** R

La tabla 6.3 de la sección 6.5 presenta el número de MABs en GT para diferentes valores de ϵ.

6.4 Algoritmo para determinar los menores árboles binarios m_k-válidos (m_k-MAB)

El algoritmo 6.3 busca en el árbol GT los MAB que son m_k-válidos (véase la definición 6.7 en la página 89). El algoritmo genera matrices parciales durante la búsqueda,

de acuerdo a la definición 6.6. Cada matriz parcial $\mathrm{A}_{\ell\times m}$ determina un Menor Subárbol Binario distinto (MSAB), mostrando como dividir los símplices en cada nivel ℓ, para formar ese subárbol en particular. Las matrices $\mathrm{A}_{(\ell+1)\times m}$ se generan a partir de $\mathrm{A}_{\ell\times m}$ (véase la ecuación (6.1) para el caso peor en el número de combinaciones) visitando nodos en el nivel $\ell+1$ del GT, siguiendo la bisección previamente determinada por la matriz parcial previa $\mathrm{A}_{\ell\times m}$. Una matriz parcial $\mathrm{A}_{\ell\times m}$ que no sea capaz de generar un MAB m_k-válido que llegue hasta el nivel $\ell+1$ será descartada de la búsqueda. La búsqueda continua hasta que no existe una matriz m_k-válida que genere un MSAB que alcance el siguiente nivel, o no exista un nivel siguiente a visitar.

Algoritmo 6.3 m_k-MAB (GT, k)

Requiere: GT: salida del algoritmo 6.1, k: para la condición m_k-válido ($m = 2^k$)

1: $\mathrm{A}_{1\times m}$:=1 ▶ Primera matriz parcial
2: Λ:= $\{\mathrm{A}_{1\times m}\}$ ▶ Conjunto de matrices parciales
3: $\Omega := \{\emptyset\}$ ▶ Conjunto de matrices finales
4: $npm := 1$ ▶ Número de matrices parciales evaluadas
5: **while** $\Lambda \neq \emptyset$ **do**
6: Extraer una matriz parcial $\mathrm{A}_{\ell\times m}$ de Λ
7: **if** $\nexists$ el nivel ℓ+1 en GT a partir de $\mathrm{A}_{\ell\times m}$ **then**
8: Almacenar $\mathrm{A}_{\ell\times m}$ en Ω
9: **else**
10: $\Gamma :=$ GenNewAs(GT, k, $\mathrm{A}_{\ell\times m}$) ▶ Generar un nuevo conjunto de matrices parciales m_k-válidas
11: Almacenar Γ en Λ
12: $npm := npm + |\Gamma|$
13: **return** Ω

El algoritmo 6.3 sigue la estructura general de un algoritmo de Ramificación y Acotación (véase el algoritmo 5.1 de la página 83), iterando sobre las matrices parciales almacenadas en Λ. La primera matriz parcial $\mathrm{A}_{\ell\times m}$ almacena los índices del primer lado mayor, pero este podría ser cualquiera de los lados porque es el símplex inicial que es regular. En la línea 10 se producen nuevas matrices parciales m_k-válidas para el nivel $\ell+1$ a partir de la matriz parcial actual $\mathrm{A}_{\ell\times m}$ (véase la definición 6.7). Por ejemplo, para un 3-símplex y $k = 0$, el algoritmo 6.3 genera en la línea 10 las ma-

trices parciales $\binom{1}{1}$, $\binom{1}{2}$ y $\binom{1}{4}$ a partir de la matriz parcial inicial $A_{1\times1}$=(1) (véase la figura 6.1). Aquellas matrices parciales que no son rechazadas y no permiten progresar en GT son almacenadas en Ω (véase la línea 8).

6.5 Resultados

Los algoritmos expuestos en las secciones anteriores han sido implementados en C++ y se han ejecutado en la máquina BullX descrita en la sección 3.4.

Tabla 6.1: A-3SNC es un ejemplo de conversión a matriz de la tabla 5.1. $A_{l\times2}$ puede extenderse a $A_{l\times4}$.

Nivel	SelecciónLM(S_i)
1	$\{1, 2\}$
2	$\{2, 3\}$
3	$\{1, 2\}$
4	$\{1, 3\}$ para i par $\{1, 2\}$ para i impar
5	$\{1, 2\}$
6	$\{1, 2\}$
7	$\{1, 2\}$ para i par $\{1, 3\}$ para i impar
8	Igual al nivel 5
9	Igual al nivel 6
10	Igual al nivel 7
...	

$$\text{A-3SNC} = A_{\ell\times2} = \begin{pmatrix} 1 & \\ 3 & 3 \\ 1 & 1 \\ 2 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 2 \\ 1 & 1 \\ 1 & 1 \\ 1 & 2 \\ \cdots & \end{pmatrix} \qquad A_{\ell\times4} = \begin{pmatrix} 1 & & & \\ 3 & 3 & & \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \\ \cdots & & & \end{pmatrix}$$

La tabla 6.1 muestra la matriz llamada A-3SNC obtenida a partir de la tabla 5.1 que muestra la secuencia de lados mayores a dividir para generar el menor número de clases para un 3-símplex. El conjunto de LMs en A-3SNC es 2-válido. Los niveles 1 a 3, y 5 y 6, son 1-válidos y los niveles 4 y 7 son 2-válidos. La matriz A-3SNC puede aplicarse para generar árboles binarios con más de 7 niveles porque los niveles 8, 9 y 10 emplean los mismos índices de LM que los niveles 5, 6 y 7, y a partir de ahí se repiten los índices del mismo modo.

La columna TamañoArbol de la tabla 6.2 muestra el tamaño menor de los arboles binarios para un 3-símplex regular obtenidos mediante el algoritmo 4.8 para varios valores de ϵ. La columna A-3SNC_en_RS() muestra el tamaño del árbol binario generado por el algoritmo (1.2) cuando se usan los índices de la tabla 6.1 para elegir el lado mayor por el que dividir los símplices. Comparando ambas columnas de la tabla 6.2

el algoritmo RS genera árboles binarios de tamaño mínimo para $\epsilon = 1/2^j$, cuando se usa la matriz A-3SNC. El uso de la matriz también devuelve resultados mínimos para otros valores de ϵ, pero existen valores de ϵ que no alcanzan un MAB. La tabla 6.2 muestra los resultados para estos valores de ϵ en **negrita**. El motivo de estos resultados es la aparición de lados mayores en el refinamiento con un tamaño diferente de $1/2^j$. En la figura, 1.7, los símplices S_5 y S_6 son ejemplos de este caso para el problema del 2-símplex.

Tabla 6.2: Tamaño del menor árbol binario (TamañoArbol()) y del árbol binario generado por RS() con la matriz A-3SNC para un 3-símplex, variando la precisión ϵ.

ϵ	TamañoArbol()	A-3SNC_en_RS()
0,8	31	31
$1/2$	47	47
0,35	**335**	**351**
$1/2^2$	351	351
0,2	1.727	1.727
$1/2^3$	2.751	2.751
0,1	13.695	13.695
$1/2^4$	21.887	21.887
0,05	**108.799**	**109.311**
$1/2^5$	174.847	174.847
0,02	**1.354.495**	**1.398.271**
$1/2^6$	1.398.271	1.398.271
0,01	**10.835.455**	**11.185.151**
$1/2^7$	11.185.151	11.185.151
0,005	**86.682.623**	**89.479.167**
$1/2^8$	89.479.167	89.479.167
0,002	715.829.247	715.829.247
$1/2^9$	715.829.247	715.829.247
0,001	1.431.658.495	1.431.658.495
$1/2^{10}$	1.431.658.495	1.431.658.495

La tabla 6.3 muestra el tamaño (|GT|) del árbol general, el número de árboles binarios de tamaño mínimo en GT, contados por el algoritmo 6.2 (MABs), el número

de matrices parciales (npm) y el número de matrices finales (nfm) evaluadas por el algoritmo 6.3, para las mismas instancias del problema mostradas en la tabla 6.2. El valor de entrada k se fija a 2 (véase la definición 6.7). El tamaño del árbol GT es mucho menor que el tamaño de uno de los correspondientes MABs porque solamente se procesa uno de los hermanos simétricos (véase la mejora 3.2 y la figura 6.1). No obstante, los resultados para valores $\epsilon < 1/2^6$ no se han podido calcular debido a los altos requisitos computacionales de los algoritmos.

El algoritmo 6.3 devuelve solo una posible matriz 4-válida, igual a A-3SNC, para los casos marcados con *check* (✓) en la tabla 6.3. En aquellos casos sin *check*, el algoritmo no ha encontrado ninguna matriz 4-válida. La matriz A-3SNC es de hecho 2-válida, pero es también 2^k-válida para un 3-símplex y $k \geq 1$ (véase la propiedad 6.2).

Tabla 6.3: Resultados de los algoritmos 6.1, 6.2 y 6.3 para un 3-símplex y $k = 2$.

ϵ	GT	MABs	npm	nfm	A-3SNC
0,8	9	1	5	1	✓
$1/2$	11	1	5	1	✓
0,35	111	16	17	0	
$1/2^2$	41	1	8	1	✓
0,2	137	9	10	1	✓
$1/2^3$	173	9	11	1	✓
0,1	617	59.049	13	1	✓
$1/2^4$	785	59.049	14	1	✓
0,05	180.708	$1{,}6{\cdot}10^{186}$	10	0	
$1/2^5$	3.665	$1{,}4{\cdot}10^{18}$	17	1	✓
0,02	34.913	$4{,}1{\cdot}10^{203}$	17	0	
$1/2^6$	17.273	$8{,}6{\cdot}10^{63}$	20	1	✓

La menor matriz posible sería 1-válida con todos los índices de LM iguales para cada nivel. La matriz A-3SNC representa el único conjunto 2-válido. Por lo tanto, no existe otra matriz válida con un menor valor de k con el reordenamiento de vértices aplicado en el algoritmo 1.3. Los resultados muestran que A-3SNC es la matriz m_k-válida con el menor valor k.

El uso de la matriz A-3SNC necesita el mismo método de reordenación de vértices para cada símplex, de acuerdo al algoritmo 1.3. El coste computacional de esta reordenación es mucho menor que el calculo de las distancias entre vértices, el cual aumenta conforme se incrementa n.

El problema análogo al abordado en este estudio es el de encontrar el menor número de reordenaciones de vértices en los símplices de forma que el mismo índice de lado mayor sea el lado a biseccionar en cualquier nodo del MAB. Este enfoque parece ser tan complejo como el abordado aquí. En el peor de los casos, sería necesario aportar una reordenación por cada símplex. Sin embargo, podría ser un enfoque interesante para abordar en un trabajo futuro.

6.6 Conclusiones

Para dar respuesta a la pregunta 1.6, en este capítulo se han estudiado varios algoritmos que conjuntamente permiten la búsqueda de matrices con una repetición alta de índices de lados mayores a dividir por nivel del árbol binario. Se ha podido comprobar que la secuencia presentada en el apartado 5.3 ([26]) es la única 2-válida y 4-válida (ver definición 6.7 y propiedad 6.2). La complejidad combinatoria del problema tan solo nos ha permitido obtener resultados para el 3-símplex y $\epsilon \geq 1/2^6$. La investigación y aplicación de los algoritmos para el problema cuando la dimensión es mayor a 4 o $\epsilon \leq 1/2^6$ requiere el uso de computación paralela debido a los requisitos computacionales y de memoria para producir una matriz 4-válida para una precisión pequeña que permita dividir cada lado inicial varias veces.

Capítulo

7

Conclusiones y trabajo futuro

En este capítulo se resumen las principales conclusiones que se obtienen tras el trabajo realizado y se sugieren futuras líneas de investigación que pueden ser de interés.

7.1 Conclusiones

En esta tesis se ha investigado el proceso de bisección iterativa de un n-símplex regular utilizando el método de bisección por el lado mayor. El problema que aparece es que para símplices de dimensión mayor que tres, la selección del lado mayor no es única, es decir, aparecen varios lados mayores que pueden ser divididos.

Tradicionalmente, por comodidad, se ha seleccionado el primer lado mayor según la ordenación de los vértices del símplex. El problema es que este método de selección del lado mayor genera un árbol binario con un número de símplices muy alto. Para reducir el número de símplices que se generan, se han desarrollado varias heurísticas de selección de lado mayor a dividir. Todas ellas generan árboles de menor tamaño que la selección del primer lado mayor, pero dependiendo de la dimensión del problema, la heurística que obtiene un árbol de menor tamaño varía. Además, utilizando cualquiera de las heurísticas presentadas, el tamaño del árbol obtenido no es el menor para $n > 3$. Por lo tanto, encontrar una heurística que consiga el menor árbol es una cuestión todavía sin resolver.

Se han desarrollado algoritmos secuenciales para obtener el tamaño del menor árbol binario generado mediante BLM de un símplex regular. Debido a la complejidad del problema que se está tratando de resolver, el coste computacional de los algoritmos diseñados es muy alto. Se han desarrollado estrategias para reducir la búsqueda y gracias a ellas se ha conseguido reducir el coste computacional de los algoritmos. Se

ha diseñado un algoritmo recursivo y otro iterativo con la finalidad de poder estudiar las ventajas e inconvenientes de estos dos enfoques de programación. Los algoritmos se han ejecutado para la resolución de diferentes instancias del problema en distintas máquinas. La versión recursiva consume un menor tiempo de ejecución para resolver las instancias del problema planteadas. Dado el alto requerimiento de memoria de los algoritmos, se observa como el uso de diferentes gestores de memoria dinámica afecta en gran medida a los tiempos de ejecución de los algoritmos. De entre todos los gestores evaluados, se ha comprobado experimentalmente como Lockless Allocator reduce considerablemente el tiempo de resolución del problema, independientemente de la instancia del problema o del método de resolución usado.

La obtención del tamaño del menor árbol binario que aparece durante el proceso de bisección iterativa de un símplex, es una tarea computacionalmente muy costosa. Para llevarla a cabo, se han desarrollado dos algoritmos paralelos que hacen uso de un número fijo de hebras (T3S y TBB) y dos con un número dinámico de hebras (RPTH e IPTH). A partir de los experimentos realizados, la conclusión es que los algoritmos TBB (algoritmo 4.5) e IPTH (algoritmo 4.8) son los que obtienen los resultados de una manera más eficiente. TBB es el algoritmo que mejor se comporta en problemas de dimensión menor que 5, debido a que no consume la memoria de las máquinas utilizadas. Sin embargo, debido a su alto requerimiento de memoria, para dimensiones mayores, su comportamiento no es eficiente y es la versión IPTH la que obtiene los mejores resultados, aunque su buen speedup no es cercano al lineal.

Para hacer una estimación del tamaño de un subárbol a partir de un nodo, el tener un árbol con el menor número de formas de símplices reduce el número de posibles subárboles y facilita el cálculo de una estimación de su tamaño. Para determinar el número mínimo de clases de símplices en el proceso de refinamiento de un símplex se ha desarrollado un algoritmo B&B que comprueba todas las clases de todos los posibles árboles binarios que se generan dependiendo del lado mayor seleccionado. Con dicho algoritmo se ha obtenido que el número mínimo de clases que aparecen durante el proceso de refinamiento de un 3-símplex es 8 [23]. Para dimensiones mayores a cuatro, debido al alto coste computacional del algoritmo, se requiere el uso de computación de altas prestaciones para poder obtener una solución en un tiempo razonable.

La investigación ha concluido desarrollando una serie de algoritmos que conjuntamente, buscan la secuencia recurrente de lados mayores a dividir y de menor tamaño

para obtener el árbol binario de menor tamaño. Se ha podido comprobar que la secuencia presentada en el apartado 5.3 ([26]) es la única 2-válida y 4-válida (ver definición 6.7 y propiedad 6.2). La complejidad combinatoria del problema tan solo nos ha permitido obtener resultados para el 3-símplex y $\epsilon \geq 1/2^6$. La investigación y aplicación de los algoritmos para el problema cuando la dimensión es mayor a 4 o $\epsilon \leq 1/2^6$ requiere el uso de computación paralela debido a los requisitos computacionales y de memoria para producir una matriz 4-válida para una precisión pequeña que permita dividir cada lado inicial varias veces.

7.2 Trabajo futuro

Como primer punto a resolver en un futuro se tiene la mejora de los algoritmos paralelos que calculan el tamaño del menor árbol binario. Para la mejora de IPTH, se plantea estudiar qué hebra ha de crear nuevas hebras y qué carga de trabajo que tiene que ir asociada a la nueva hebra. También se plantea la inclusión del trabajo propuesto en [69] en la versión IPTH. Esta mejora conseguirá que el número máximo de hebras que se puede generar varíe de manera dinámica según las condiciones de la máquina en la que se esté ejecutando el algoritmo.

Para la mejora de IPTH o TBB, se propone como trabajo futuro, obtener un algoritmo que compruebe de manera sencilla si los pares de símplices generados en la división de un mismo símplex por lados diferentes son semejantes en forma. En caso afirmativo, sólo una pareja debería ser procesada y por tanto, el coste computacional de procesado del símplex se vería reducido en gran medida. Como ejemplo, tenemos un 3-símplex, que en su primera división genera un símplex con tres lados mayores. Tres lados mayores dan lugar a tres parejas, seis símplices a analizar, en una pareja dos símplices son gemelos y solo uno de ellos será analizado. Las otras dos parejas tienen forma semejante y sólo una de esas parejas debería ser analizada. Por lo tanto, en vez de tener que analizar los subárboles generados por seis símplices, sólo hay que analizar los subárboles generados por tres.

Conocido el tamaño de árbol mínimo para una determinada dimensión y precisión aparecen dos retos. El primero es, para una dimensión de símplices mayor a 4, obtener una secuencia de lados mayores a dividir que nos lleven a la obtención de dicho árbol. De este modo, se reduciría en gran medida el coste computacional asociado al cálculo

del lado mayor a dividir en el proceso de división de un n-símplex. El segundo reto es cómo realizar una previsión de la carga computacional pendiente, cuando se ejecuta un algoritmo de B&B y se conoce el tamaño de árbol mínimo.

Otra futura línea de investigación es el estudio de las clases de símplices que se generan en el refinamiento por el lado mayor. Se aplicará computación paralela para obtener el árbol con menor número de clases cuando la dimensión del símplex a refinar es mayor que 4. Se estudiará si, como sucede para el caso del 3-símplex, el árbol de tamaño mínimo y el árbol con menor número de clases coinciden.

Apéndice

A

Publicaciones que han contribuido a esta tesis

Este apéndice contiene la lista de publicaciones (extraída de la bibliografía) derivadas de la investigación desarrollada para la elaboración de esta tesis.

A.1 Publicaciones en revistas internacionales

[26] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2015). On the minimim number of simplex shapes in longest edge bisection refinement of a regular n-simplex. *Informatica*, DOI: 10.15388/Informatica.2015.36. **26**, 17–32
Impact factor JCR 2014: 0.873. Journal ranking: 114/255 in *Mathematics, Applied*.

[68] Salmerón, J.M.G., **Guillermo Aparicio**, Casado, L.G., García, I., Hendrix, E.M.T. & G.-Tóth, B. (2015). Generating a smallest binary tree by proper selection of the longest edges to bisect in a unit simplex refinement. DOI: 10.1007/s10878-015-9970-y. *Journal of Combinatorial Optimization*, in press
Impact factor JCR 2014: 0.939. Journal ranking: 101/255 in *Mathematics, Applied*. Journal ranking: 75/102 in *Computer science, intercisplinary applications*.

[28] **Guillermo Aparicio**, Salmerón, J.M.G., Casado, L.G., Asenjo, R. & Hendrix, E.M.T. (2015). Parallel algorithms for computing the size of the smallest binary tree in unit simplex refinement. *Parallel Computing, Systems & Applications*, submitted.

Impact factor JCR 2014: 1.511. Journal ranking: 23/102 in *Computer Science, Theory and Methods*.

A.2 Publicaciones en proceedings de conferencias internacionales con DOI

[24] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2014). Heuristics to reduce the number of simplices in longest edge bisection refinement of a regular n-simplex. In *Computational Science and Its Applications – ICCSA 2014*, vol. 8580 of *Lecture Notes in Computer Science*, 115–125, DOI: 10.1007/978-3-319-09129-7_9. Springer International Publishing

[23] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2013). On computational aspects of a regular n-simplex bisection. In *Proceedings of the Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 513–518, DOI: 10.1109/3PGCIC.2013.88. IEEE Computer Society, Compiegne, France

A.3 Publicaciones en proceedings de otras conferencias internacionales

[25] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2014). On the minimim number of simplex shapes in longest edge bisection refinement of a unit simplex. In *Proceedings of the XII Global Optimization Workshop. Mathematical and Applied Global Optimization (MAGO 2014)*, 65–68

[4] Casado, L.G., **Guillermo Aparicio**, G.-Tóth, B., Hendrix, E.M.T. & García, I. (2013). On simplex longest edge bisection for solving blending problems. In *Book of Abstracts of the International Conference on Continuous Optimization (ICCOPT'13)*, 85, University Nova Lisboa, Portugal

A.4 Publicaciones en proceedings de conferencias nacionales

[27] **Guillermo Aparicio**, Salmerón, J.M.G., Casado, L.G., Asenjo, R., García, I. & Hendrix, E.M.T. (2015). Algoritmos paralelos de memoria compartida que determinan el menor tamaño de un árbol binario al refinar un simplex regular. 387–392, Córdoba, *Actas XXVI Jornadas de Paralelismo*

[67] Salmerón, J.M.G., Herrera, J.F.R., **Guillermo Aparicio**, Casado, L.G., García, I. & Hendrix, E.M.T. (2014). Estrategias paralelas para obtener el tamaño del árbol mínimo en la división por el lado mayor de un símplice regular. 29–35, Valladolid, *Actas XXV Jornadas de Paralelismo*

[22] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2013). Aspectos computacionales en la bisección de un n-simplex regular. 396–401, Madrid, *Actas XXIV Jornadas de Paralelismo*

Bibliografía

[1] Bomze, I., Csendes, T., Horst, R. & Pardalos, P., eds. (1997). *Developments in Global Optimization*, vol. 18 of *Noncovex optimization and its applications*. DOI: 10.1007/978-1-4757-2600-8. Kluwer Academic Publishers.

[2] Casado, L.G., Martínez, J., García, I. & Hendrix, E.M.T. (2008). Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods & Software*, DOI: 10.1080/10556780802086300. **23**, 689–701.

[3] Casado, L.G., García, I., G.-Tóth, B. & Hendrix, E.M.T. (2011). On determining the cover of a simplex by spheres centered at its vertices. *Journal of Global Optimization*, DOI: 10.1007/s10898-010-9524-x. **50**, 645–655.

[4] Casado, L.G., **Guillermo Aparicio**, G.-Tóth, B., Hendrix, E.M.T. & García, I. (2013). On simplex longest edge bisection for solving blending problems. In *Book of Abstracts of the International Conference on Continuous Optimization (ICCOPT'13)*, 85, University Nova Lisboa, Portugal.

[5] Conti, P., Hitschfeld, N. & Fichtner, W. (1991). Omega;-an octree-based mixed element grid allocator for the simulation of complex 3-d device structures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, DOI: 10.1109/43.88919. **10**, 1231–1241.

[6] Contreras, G. & Martonosi, M. (2008). Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, DOI: 10.1109/IISWC.2008.4636091. 57–66.

[7] Cormen, T., Leiseron, C. & Rivest, R. (1990). *Introduction to Algorithms*. The MIT Press.

[8] Corrêa, R. & Ferreira, A. (1995). On the effectiveness of synchronous branch-and-bound algorithms. *Parallel Processing Letters*, DOI: 10.1142/S0129626495000357. **5**, 375–386.

[9] Corrêa, R. & Ferreira, A. (1996). Parallel best-first branch and bound in discrete optimization: a framework. In A. Ferreira & P. Pardalos, eds., *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*, 145–170, DOI: 10.1007/BFb0027122. Springer LNCS 1054.

[10] de Bruin, A., Kindervater, G. & Trienekens, H. (1996). Towards an abstract parallel branch and bound machine. In A. Ferreira & P. Pardalos, eds., *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*, 145–170, DOI: 10.1007/BFb0027121. Springer LNCS 1054.

[11] Diderich, C. & Gengler, M. (1996). Solving traveling salesman problems using a parallel synchronized branch and bound algorithm. DOI: 10.1007/3-540-61142-8_607. **1067**, 633–638.

[12] Dompierre, J., Labbé, P., Guibault, F. & Camarero, R. (1998). Proposal of benchmarks for 3d unstructured tetrahedral mesh optimization. In *In Proceedings of the 7th International Meshing RoundTable'98*, 459–478.

[13] Dompierre, J., Vallet, M.G., Labbé, P. & Guibault, F. (2005). An analysis of simplex shape measures for anisotropic meshes. *Computer Methods in Applied Mechanics and Engineering*, DOI: 10.1016/j.cma.2004.11.018. **194**, 4895 – 4914, unstructured Mesh Generation.

[14] Estrada, J.F.S., Casado, L.G. & García, I. (2011). Adaptive parallel interval global optimization algorithms based on their performance for non-dedicated multicore architectures. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, DOI: 10.1109/PDP.2011.54. 252–256.

[15] Estrada, J.F.S., Casado, L.G., García, I. & Hendrix, E.M.T. (2012). Performance driven cooperation between kernel and auto-tuning multi-threaded interval b&b applications. In B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. Rocha, D. Taniar & B. Apduhan, eds., *Computational Science and Its Applications – ICCSA 2012*, vol. 7333 of *Lecture Notes in Computer Science*, 57–70, DOI: 10.1007/978-3-642-31125-3_5. Springer Berlin Heidelberg.

[16] Estrada, J.F.S., Casado, L.G. & García, I. (2015). On sleeping threads to maintain the efficiency in auto-tuned shared memory spmd applications with dynamic number of threads. In *Proceedings of the 2010 International Conference on Mathematical Methods in Science and Engineering (CMMSE'15)*, 1004–1010, Rota, Cádiz (Spain).

[17] Floudas, C. & Pardalos, P. (1990). *A collection Test Problems for Constrained Global Optimization Algorithms*, vol. 455 of *Lectures Notes in Computer Science*. DOI: 10.1007/3-540-53032-0. Springer.

[18] Floudas, C. & Pardalos, P. (1992). *Recent Advances in Global Optimization*. Princeton series in Computer Science, Princeton.

[19] Garey, M. & Johnson, D. (1979). *Computer and Intractability*. W.H. Freeman and Company.

[20] Gengler, M. & Coray, G. (1994). A parallel best-first B&B algorithm and its axiomatization. vol. 2, DOI: 10.1109/HICSS.1993.284102. 61–80.

[21] Grossmann, I. (1996). *Global Optimization in Engineering Design*. DOI: 10.1007/978-1-4757-5331-8. Kluwer Academic Publishers.

[22] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2013). Aspectos computacionales en la bisección de un n-simplex regular. 396–401, Madrid, *Actas XXIV Jornadas de Paralelismo*.

[23] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2013). On computational aspects of a regular n-simplex bisection. In *Proceedings of the Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 513–518, DOI: 10.1109/3PGCIC.2013.88. IEEE Computer Society, Compiegne, France.

[24] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2014). Heuristics to reduce the number of simplices in longest edge bisection refinement of a regular n-simplex. In *Computational Science and Its Applications – ICCSA 2014*, vol. 8580 of *Lecture Notes in Computer Science*, 115–125, DOI: 10.1007/978-3-319-09129-7_9. Springer International Publishing.

[25] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2014). On the minimim number of simplex shapes in longest edge bisection refinement of a unit simplex. In *Proceedings of the XII Global Optimization Workshop. Mathematical and Applied Global Optimization (MAGO 2014)*, 65–68.

[26] **Guillermo Aparicio**, Casado, L.G., Hendrix, E.M.T., García, I. & G.-Tóth, B. (2015). On the minimim number of simplex shapes in longest edge bisection refinement of a regular n-simplex. *Informatica*, DOI: 10.15388/Informatica.2015.36. **26**, 17–32.

[27] **Guillermo Aparicio**, Salmerón, J.M.G., Casado, L.G., Asenjo, R., García, I. & Hendrix, E.M.T. (2015). Algoritmos paralelos de memoria compartida que determinan el menor tamaño de un árbol binario al refinar un simplex regular. 387–392, Córdoba, *Actas XXVI Jornadas de Paralelismo*.

[28] **Guillermo Aparicio**, Salmerón, J.M.G., Casado, L.G., Asenjo, R. & Hendrix, E.M.T. (2015). Parallel algorithms for computing the size of the smallest binary tree in unit simplex refinement. *Parallel Computing, Systems & Applications*, submitted.

[29] Hannukainen, A., Korotov, S. & Křížek, M. (2014). On numerical regularity of the face-to-face longest-edge bisection algorithm for tetrahedral partitions. *Science of Computer Programming*, DOI: 10.1016/j.scico.2013.05.002. **90**, 34–41.

[30] Hendrix, E.M.T. & G.-Tóth, B. (2010). *Introduction to Nonlinear and Global Optimization*, vol. 37 of *Springer Optimization and Its Applications*. DOI: 10.1007/978-0-387-88670-1. Springer-Verlag New York.

[31] Herrera, J., Casado, L.G., Paulavičius, R., Žilinskas, J. & Hendrix, E.M.T. (2013). On a hybrid MPI-Pthread approach for simplicial branch-and-bound. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS'13)*, 1764–1770, DOI: 10.1109/IPDPSW.2013.178. Boston, Massachusetts USA.

[32] Herrera, J.F.R., Casado, L.G., Hendrix, E.M.T. & García, I. (2013). A threaded approach of the quadratic bi-blending algorithm. *Journal of Supercomputing*, DOI: 10.1007/s11227-012-0783-9. **64**, 38–48.

[33] Herrera, J.F.R., Casado, L.G., Paulavičius, R., Žilinskas, J. & Hendrix, E.M.T. (2013). On a hybrid MPI-Pthread approach for simplicial branch-and-bound. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS'13)*, 1764–1770, DOI: 10.1109/IPDPSW.2013.178. Boston, Massachusetts USA.

[34] Herrera, J.F.R., Casado, L.G., Hendrix, E.M.T. & García, I. (2014). On simplicial longest edge bisection in lipschitz global optimization. In *Computational Science and Its Applications – ICCSA 2014*, vol. 8580, 104–114, DOI: 10.1007/978-3-319-09129-7. Springer International Publishing.

[35] Herrera, J.F.R., Casado, L.G., Hendrix, E.M.T. & García, I. (2014). Pareto optimality and robustness in bi-blending problems. *TOP*, DOI: 10.1007/s11750-012-0253-9. **22**, 254–273.

[36] Horst, R. & Pardalos, P., eds. (1995). *Handbook of Global Optimization*, vol. 2 of *Noncovex Optimization and its Applications*. DOI: 10.1007/978-1-4757-5362-2. Kluwer Academic Publishers, Dordrecht, Holland.

[37] Ibaraki, T. (1976). Theoretical comparisons of search strategies in branch and bound algorithms. *Int. J. Comput. Inform. Sci.*, DOI: 10.1007/BF00998631. **5**, 315–344.

[38] Ibaraki, T. (1976). Theoretical comparisons of search strategies in branch-and-bound algorithms. *International Journal of Computer & Information Sciences*, DOI: 10.1007/BF00998631. **5**, 315–344.

[39] Ibaraki, T. (1977). On the computational efficiency of branch and bound algorithms. *J. Oper. Res. Soc.*, **20**, 16–35.

[40] Ibaraki, T. (1977). The power of dominance relations in branch and bound algorithms. *J. Assoc. Comput. Match.*, DOI: 10.1145/322003.322010. **24**, 264–279.

[41] Ibaraki, T. (1987). *Enumerative approaches to combinatorial optimization*, vol. I and II. J.C. BALTZER AG, Basel-Switzerland.

[42] Kac, M. & Logan, J. (1976). Fluctuation phenomena. In *Fluctuation Phenomena*, E.W. Montroll and J.L. Lebowitz, North-Holland, Ámsterdam.

[43] Kearfott, B. (1978). A proof of convergence and an error bound for the method of bisection in $\mathbf{R}^n$. *Journal of Mathematical Computing*, DOI: 10.2307/2006341. **32**, 1147–1153.

[44] Kukanov, A. & Voss, M.J. (2007). The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, DOI: 10.1535/itj.1104. **11**, 309–322.

[45] Kumar, V. & Kanal, L. (1983). A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, DOI: 10.1016/S0004-3702(83)80009-5. **21**, 179–198.

[46] Lawer, E. & Wood, D. (1966). Branch and bound methods: A survey. *Operation Research*, DOI: 10.1287/opre.14.4.699. **14**, 699–719.

[47] Little, J., Murty, K., Sweeney, D. & Karel, C. (1963). An algorithm for the traveling salesman problem. *Operation Research*, DOI: 10.1287/opre.11.6.972. **11**, 972–989.

[48] Lüling, R., Monien, B., Reinefeld, A. & Tschöke, S. (1996). Mapping tree-structured combinatorial optimization problems onto parallel computers. In A. Ferreira & P. Pardalos, eds., *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*, DOI: 10.1007/BFb0027120. Springer LNCS 1054.

[49] Mitchell, W.F. (1989). A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Softw.*, DOI: 10.1145/76909.76912. **15**, 326–347.

[50] Mitten, L.G. (1970). Branch and bound methods: general formulation and properties. *Operation Research*, **18**, 24–34.

[51] Nau, D., Kumar, V. & Kanal, L. (1984). General branch and bound and its relation to A* and AO*. *Artificial Intelligence*, DOI: 10.1016/0004-3702(84)90004-3. **23**, 29–58.

[52] Nelson, E. (1985). Quantum fluctuations. In *Quantum Fluctuations*, Princeton University Press, Princeton.

[53] Neumaier, A. (2004). Complete search in continuous global optimization and constraint. *Acta Numerica*, **13**, 271–369.

[54] Pardalos, P. & Rosen, J. (1987). *Constrained Global Optimization: Algorithms and Applications*. Lectures Notes in Computer Science, DOI: 10.1007/BFb0000035. Springer.

[55] Pardalos, P. & Schnitger, G. (1988). Checking local optimality in constrained quadratic programming is NP-hard. *Operations Research Letters*, DOI: 10.1016/0167-6377(88)90049-1. **7**, 33–35.

[56] Pardalos, P. & Vavasis, S. (1991). Quadratic programation with one negative eigenvalue is NP-hard. *Journal of Global Optimization*, DOI: 10.1007/BF00120662. **1**, 15–22.

[57] Parthasarathy, V., Graichen, C. & Hathaway, A. (1994). A comparison of tetrahedron quality measures. *Finite Elements in Analysis and Design*, DOI: 10.1016/0168-874X(94)90033-7. **15**, 255 – 261.

[58] Paulavičius, R. & Žilinskas, J. (2014). *Simplicial Global Optimization*. DOI: 10.1007/978-1-4614-9093-7. Springer-Verlag New York.

[59] Paulavičius, R. & Žilinskas, J. (2014). Simplicial lipschitz optimization without the lipschitz constant. *Journal of Global Optimization*, DOI: 10.1007/s10898-013-0089-3. **59**, 23–40.

[60] Paulavičius, R., Žilinskas, J. & Grothey, A. (2011). Parallel branch and bound for global optimization with combination of lipschitz bounds. *Optimization Methods and Software*, DOI: 10.1080/10556788.2010.551537. **26**, 487–498.

[61] Pintér, J. (1996). Continuous global optimization software: A brief review. *Newsletter of the Mathematical Programming Society*, **52**, 1–8.

[62] Pintér, J.D. (1996). *Global Optimization in Action*, vol. 6 of *Noncovex optimization and its applications*. DOI: 10.1007/978-1-4757-2502-5. Kluwer Academic Publishers, Dordrecht, The Netherland.

[63] Plaza, A. & Carey, G.F. (2002). Explicación geométrica de una medida de forma de símplices n-dimensional. *Revista Internacional de Métodos Numéricos en la Ingeniería*, **18**, 475–480.

[64] Rauber, T. & Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. DOI: 10.1007/978-3-642-37801-0. Springer.

[65] Reinders, J. (2007). *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly.

[66] Rossman, M., Twery, R. & Stone, F. (1958). A solution to the traveling salesman problem by combinatorial programming. Chicago.

[67] Salmerón, J.M.G., Herrera, J.F.R., **Guillermo Aparicio**, Casado, L.G., García, I. & Hendrix, E.M.T. (2014). Estrategias paralelas para obtener el tamaño del árbol mínimo en la división por el lado mayor de un símplice regular. 29–35, Valladolid, *Actas XXV Jornadas de Paralelismo*.

[68] Salmerón, J.M.G., **Guillermo Aparicio**, Casado, L.G., García, I., Hendrix, E.M.T. & G.-Tóth, B. (2015). Generating a smallest binary tree by proper selection of the longest edges to bisect in a unit simplex refinement. DOI: 10.1007/s10878-015-9970-y. *Journal of Combinatorial Optimization*, in press.

[69] Sanjuan-Estrada, J., Casado, L.G., García, I. & Hendrix, E.M.T. (2012). Performance driven cooperation between kernel and auto-tuning multi-threaded interval B&B applications. DOI: 10.1007/978-3-642-31125-3_5. **7333**, 57–70.

[70] Sanjuan-Estrada, J.F., Casado, L.G. & García, I. (2011). Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures. *Journal of Supercomputing*, DOI: 10.1007/s11227-011-0594-4. **58**, 376–384.

[71] Schoen, F. (1991). Stochastic techniques for global optimization: A survey of recent advances. *Journal of Global Optimization*, DOI: 10.1007/BF00119932. **11**, 207–228.

[72] Stynes, M. (1980). On faster convergence of the bisection method for all triangles. *Mathematics of Computation*, DOI: 10.2307/2006385. **35**, pp. 1195–1201.

[73] Törn, A. & Žilinskas, A. (1989). *Global Optimization*, vol. 3350. DOI: 10.1007/3-540-50871-6. Springer-Verlag, Berlin, Germany.

[74] Tousimojarad, A. & Vanderbauwhede, W. (2014). Comparison of three popular parallel programming models on the Intel Xeon Phi. In *Euro-Par 2014: Parallel Processing Workshops*, vol. 8806 of *Lecture Notes in Computer Science*, 314–325, DOI: 10.1007/978-3-319-14313-2. Springer International Publishing.

[75] Trienekens, H. (1990). *Parallel Branch and Bound Algorithms*. Ph.D. thesis, Erasmus University, Rotterdam.

[76] Tschöke, S., Lüling, R. & Monien, B. (1995). Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *9th International Parallel Processing Symposium (IPPS '95)*, 182–189, DOI: 10.1109/IPPS.1995.395930. ta Barbara.

[77] Žilinskas, A. & Žilinskas, J. (2013). A hybrid global optimization algorithm for non-linear least squares regression. *Journal of Global Optimization*, DOI: 10.1007/s10898-011-9840-9. **56**, 265–277.

[78] Žilinskas, J. & Dür, M. (2011). Depth-first simplicial partition for copositivity detection, with an application to maxclique. *Optimization Methods and Software*, DOI: 10.1080/10556788.2010.544310. **26**, 499–510.

[79] Weicker, R.P. (1984). Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, DOI: 10.1145/358274.358283. **27**, 1013–1030.

[80] Whitehead, J.H.C. (1940). On c1-complexes. *Annals of Mathematics*, DOI: 10.2307/1968861. **41**, pp. 809–824.