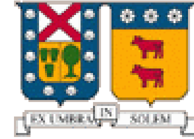
Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso – Chile



PROGRAMACIÓN CON RESTRICCIONES PARA EL TRATAMIENTO DE INCERTIDUMBRE EN CSP NUMÉRICOS

Tesis presentada como requerimiento parcial
para optar al grado académico de

MAGISTER EN INGENIERÍA INFORMÁTICA

y al título profesional de

INGENIERO CIVIL EN INFORMÁTICA

por

Carlos Grandón Espinoza

Comisión Evaluadora:
Dra. María Cristina Riff (Guía, UTFSM)
Dr. Carlos Castro (UTFSM)
Dr. Eric Monfroy (IRIN, Nantes, France)

ENERO 2004

Universidad Técnica Federico Santa María
Departamento de Informática
Valparaíso – Chile

TITULO DE LA TESIS:

PROGRAMACIÓN CON RESTRICCIONES PARA EL TRATAMIENTO DE INCERTIDUMBRE EN CSP NUMÉRICOS

AUTOR:

CARLOS GRANDÓN ESPINOZA

Tesis presentada como requerimiento parcial para optar al grado académico de **Magister en Ingeniería Informática** y al título profesional de **Ingeniero Civil en Informática** de la Universidad Técnica Federico Santa María.

Dra. María Cristina Riff ______________________ Profesor Guía

Dr. Carlos Castro ______________________ Profesor Correferente

Dr. Eric Monfroy ______________________ Profesor Externo

Enero 2004.
Valparaíso, Chile.

A Marcela y mis hijos.

Agradecimientos

Es difícil restringir a una página todos los agradecimientos que por una u otra razón merecen aquellas personas que hicieron posible la realización de esta Tesis. En primer lugar están, como siempre, mis padres. Con sus virtudes y defectos han sabido apoyarme y guiarme, a través de sus experiencias y vivencias, que muchas veces han sido de ayuda invaluable. Mis más sinceros agradecimientos, por creer en mí, aún en aquellos momentos en que yo no creí.

A Marcela y mis hijos, a quienes dedico con mucho amor esta Tesis, por todos los años que han pasado a mi lado. Mis esfuerzos y sacrificios ha estado acompañados, sin duda alguna, por sus esfuerzos y sacrificios. Gracias por permanecer conmigo, por sobre todas las dificultades que hemos vivido.

A mis hermanos, Andrea, Nicolás y Diego, porque gracias a su fuerza y su apoyo, he podido superar etapas muy dolorosas, que me han servido para aprender y mejorar muchos aspectos de mi vida. Les agradezco por ser ustedes, mis hermanos.

A mis suegros, por el apoyo que le han dado a Marcela y a mis hijos durante todos estos años.

A mis amigos más cercanos, que a pesar de ser pocos, han sido de mucha ayuda y apoyo. Entre ellos Michael, porque siempre he sentido que puedo confiar en ti, y porque nunca nos has dado la espalda.

A mi profesora guía, Dra. María Cristina Riff, porque gracias a ella he adquirido conocimientos y experiencia que me han ayudado a crecer, no solo en el plano académico, sino también en el personal. Gracias por el apoyo y la confianza que depositó en mí, han sido de gran ayuda.

Finalmente deseo agradecer a todo el cuerpo docente del departamento, y además, a tres personas muy especiales, que con su alegría y diligencia, hacen más grado y llevadero los difíciles días de muchos estudiantes. Me refiero a Lidia, Carolina e Ignacio. Para ustedes, que apoyan enormemente desde el anonimato, les dedico mis sinceros agradecimientos.

Valparaíso, Chile Carlos Grandón Espinoza
Enero 12, 2004

Resumen

La incertidumbre en Problemas de Satisfacción de Restricciones involucra un cierto grado de ignorancia en la determinación de uno o más parámetros al modelar un problema. Por ejemplo, los problemas que involucran restricciones de distancia pueden tener incertidumbre en la medida de la distancia. Esta incertidumbre produce un intervalo de tolerancia en las soluciones del problema, y por consiguiente, varios conjuntos continuos de posibles soluciones.

En este trabajo se estudia una metodología basada en Programación por Restricciones para manejar la incertidumbre en problemas de ecuaciones de distancia, y de qué manera es posible extender esta metodología a un problema más general. El propósito es mejorar el algoritmo de tratamiento de incertidumbre en sistemas de ecuaciones de distancia, propuesto en [Tou02], el cual determina una aproximación interna y externa para cada conjunto solución, y extender esta metodología a un problema de satisfacción de restricciones sobre los reales más general.

Palabras Claves: Problemas de Satisfacción de Restricciones; Ecuaciones de distancia; Programación con Restricciones.

Abstract

Uncertainty in Constraint Satisfaction Problems involves a certain degree of ignorance in the measure of one or more parameters when modelling a problem. For example, problems that involve distance constraints can have uncertainty in the measure of the distance. This uncertainty produces a tolerance in the problem solutions, and therefore, several continuous sets of possible solutions.

In this work we study a methodology based in Constraint Programming for tackling uncertainty in the distance equations problems, and how we can extend this methodology to a more general problem. Our purpose is to improve the algorithm of treatment of uncertainties in the distance equations systems, proposed in [Tou02], which determines internal and external boxes for approximating solution sets, and to extend this methodology to more general Numeric Constraint Satisfaction Problems.

Keywords: Constraint Satisfaction Problems; Distance Equations Systems; Constraint Programming.

Índice de Contenidos

Agradecimientos iv

Resumen v

Abstract vi

Índice de Contenidos vii

Índice de cuadros x

Índice de figuras xi

Abreviaciones xiii

Glosario xiv

I Visión General 1

1. Introducción 2

1.1. Problemas de satisfacción de restricciones 2
1.2. Programación con Restricciones 3
1.3. Metodología actual de resolución 3
1.4. Ámbito del Trabajo 4
1.4.1. Motivación 4
1.4.2. Hipótesis 5
1.4.3. Organización de la Tesis 5

II Estado del Arte 7

2. Problemas de Satisfacción de Restricciones 8

2.1. Definiciones previas 8
2.2. Complejidad 11
2.2.1. Complejidad algorítmica 12

2.2.2. Complejidad de un problema . . . 13
2.3. Resumen del capítulo . . . 14

3. Programación con Restricciones **15**
3.1. Introducción . . . 15
3.2. Aritmética de Intervalos . . . 17
3.2.1. Definiciones en aritmética de intervalos . . . 18
3.2.2. Propiedades . . . 20
3.3. Técnicas de Consistencia . . . 20
3.3.1. Consistencia en dominios finitos . . . 21
3.3.2. Consistencia externa en dominios continuos . . . 23
3.3.3. Consistencia interna en dominios continuos . . . 26
3.4. Ejemplo gráfico de técnicas de consistencia . . . 28
3.5. Resumen del capítulo . . . 29

4. Metodología de Resolución **31**
4.1. Explicación mediante un ejemplo . . . 32
4.2. Algoritmo . . . 37
4.2.1. Cálculo de cajas interiores . . . 38
4.2.2. Cálculo de cajas exteriores . . . 39
4.3. Resumen del capítulo . . . 42

III Trabajo aplicado **43**

5. Áreas de mejoramiento **44**
5.1. Determinación cualitativa del tipo de configuración . . . 45
5.2. Influencia de la incertidumbre previa . . . 47
5.3. Relación entre los puntos de una solución . . . 48
5.4. Dificultades del algoritmo de separación de puntos . . . 49
5.5. Conclusión del capítulo . . . 51

6. Mejoras Propuestas **52**
6.1. Detección y clasificación de tipos de problemas . . . 52
6.2. Un algoritmo de separación de puntos para restricciones de distancia . . . 53
6.3. Algoritmo de separación de soluciones . . . 55
6.4. Algoritmo de análisis de soluciones . . . 57
6.5. Metodología . . . 57
6.6. Resultados preliminares . . . 58
6.6.1. Sistema de ecuaciones de distancia en el plano . . . 59
6.6.2. Determinación de la estructura de una molécula . . . 61
6.6.3. Extensión de la metodología a un problema más general . . . 64
6.7. Conclusión del capítulo . . . 68

7. Conclusiones **69**

IV Apéndices **72**

A. ILOG Concert/Solver **73**
A.1. Introducción 73
A.2. Definición de un modelo (ILOG Concert) 73
A.3. Resolución del modelo (ILOG Solver) 75
A.4. Elementos de un modelo 76
A.4.1. Tipos de Variables 76
A.4.2. Expresiones 77
A.4.3. Tipos de Restricciones 78
A.4.4. Función Objetivo 79
A.5. Manipulación de la resolución 80
A.6. Ejemplo 82
A.7. Consideraciones sobre dominios continuos 82

B. Algoritmo General **85**

Bibliografía **98**

Índice de cuadros

1. Abreviaciones utilizadas en la Tesis xiii
2. Términos de uso común en la Tesis xiv

4.1. Algoritmo de extensión de dominios i-consistentes 38
4.2. Algoritmo general de extensión de dominios 39
4.3. Función de verificación de coordenadas 40
4.4. Función de separación de puntos en espacios disjuntos 41

6.1. Algoritmo general de separación de puntos 54
6.2. Aplicación de una función de separación 55
6.3. Resultados obtenidos con el algoritmo actual 59
6.4. Resultados obtenidos con el algoritmo propuesto 60
6.5. Información de distancia entre átomos 62
6.6. Resumen de resultados obtenidos por ambos algoritmos 63
6.7. Detalle de los errores encontrados en algoritmo propuesto 63
6.8. Descripción de los principales objetos definidos 64
6.9. Resultados del sistema parábola-recta 65
6.10. Resultados para el sistema seno-parábola 67

Índice de figuras

2.1. Ejemplo de un CSP sobre dominios discretos 10
2.2. Ejemplo de un CSP sobre dominios continuos 11

3.1. Ejemplo de aplicación de aritmética de intervalos. 17
3.2. Ejemplo de un CSP no arco-consistente (A) y uno arco-consistente (B) . . . 22
3.3. Ejemplo de un CSP arco-consistente sin solución 22
3.4. Extensión de dominios consistentes en CSP 28
3.5. Relación entre la e-consistencia y la i-consistencia 29

4.1. Ejemplo de un problema de ecuaciones de distancia 32
4.2. Simplificación del problema en un plano cartesiano 33
4.3. Problema en plano cartesiano considerando incertidumbres 34
4.4. Cálculo de caja interior a través de consistencia interna 35
4.5. Separación de puntos pertenecientes a soluciones diferentes 36
4.6. Cálculo de cajas exteriores para cada solución 37
4.7. Caja exterior calculada sin separación de conjuntos solución 40

5.1. Ejemplos de configuración solución 45
5.2. Ejemplo de casos degenerados 46
5.3. Influencia de las incertidumbres previas 47
5.4. Influencia parcial en la determinación de puntos solución 48
5.5. Aplicación del algoritmo de separación de puntos 50
5.6. Cálculo de cajas exteriores a partir del espacio dividido 51

6.1. Ejemplo de aplicación de una estructura de árbol 56
6.2. Resultados del algoritmo actual, en el plano cartesiano 60
6.3. Resultados del algoritmo propuesto, en el plano cartesiano 61

6.4. Ejemplo de aplicación para una parábola y una recta 65
6.5. Ejemplo de una función senoidal y una parábola 66

Abreviaciones

Abrev.	**Significado**	**Traducción**
CSP	Constraint Satisfaction Problem	Problema de Satisfacción de Restricciones
CP	Constraint Programming	Programación con Restricciones

Cuadro 1: Abreviaciones utilizadas en la Tesis

Glosario

Término	**Explicación**
Espacio solución	Conjunto de valores que cumplen las restricciones del problema
Espacio de búsqueda	Producto cartesiano de los dominios de las variables
Solución del problema	Instanciación de las variables que cumple las restricciones del problema
Punto solución	Conjunto de una o más variables que representan una sub-solución del problema

Cuadro 2: Términos de uso común en la Tesis

Parte I

Visión General

Capítulo 1

Introducción

El trabajo presentado en esta tesis se centra principalmente en dos grandes áreas de investigación: Los problemas de satisfacción de restricciones y las técnicas basadas en la programación con restricciones para su resolución. En cuanto a los problemas, el estudio se centra en la resolución de los CSP con variables que poseen *dominios reales*. Con respecto a las técnicas, se consideran las *técnicas de consistencia* y *análisis de intervalos* pertenecientes al área de la programación con restricciones.

1.1. Problemas de satisfacción de restricciones

Los problemas de satisfacción de restricciones tienen una gran importancia en diversas áreas del quehacer humano. Muchos de los problemas de la vida cotidiana pueden ser modelados a través de un conjunto de entidades y sus relaciones particulares. Una subcategoría de esta clase de problemas la constituyen los CSP con variables continuas, y más específicamente, aquellos que manejan intervalos. Esta clase de problemas poseen una gran importancia en áreas como la biología molecular [KB99, Bac98]. Uno de los problemas fundamentales en esta área consiste en determinar la estructura molecular de una proteína a partir de la información de sus aminoácidos constituyentes y las relaciones de distancia entre ellos.

Debido a que las mediciones de la distancia entre los aminoácidos de una proteína se obtienen a través de métodos experimentales, se tiene un cierto grado de incertidumbre en la medición. Por otro lado, basándose en estudios de fuerzas de atracción entre las moléculas, es posible determinar límites para estas distancias y de esta forma acotar las posibles soluciones

del problema. Dada la incertidumbre asociada, no es posible entregar una solución exacta al problema, por lo que se hace necesario estudiar técnicas de tratamiento de incertidumbre para aproximar sus soluciones. El estudio de estas técnicas y su aplicabilidad en sistemas de ecuaciones sobre los reales es la principal motivación del trabajo de esta tesis.

1.2. Programación con Restricciones

La programación con restricciones se basa en el modelamiento y resolución con restricciones [MS98, Tho93]. Dado un problema descrito a través de variables, donde cada variable posee un dominio asociado, compuesto por un conjunto de potenciales valores y un conjunto de restricciones que representan las diferentes relaciones entre las variables que deben ser satisfechas para resolver el problema, la idea principal de la CP es usar el conocimiento de las restricciones para eliminar del dominio de las variables todos aquellos valores que no pueden formar parte de una solución al problema. De esta forma, es posible resolver un problema reduciendo los dominios de sus variables hasta conseguir aproximaciones muy cercana al valor solución, o reducir el dominio de una o más variables hasta eliminar todos sus posibles valores (en cuyo caso el problema no tiene solución).

1.3. Metodología actual de resolución

El problema de ecuaciones de distancia con incertidumbre puede ser descrito de la siguiente forma: Sea $E = R^n$, con $n > 1$, un espacio n-dimensional con base en los reales. Sea $P_1, \ldots, P_m$ un conjunto de m puntos en el espacio E, con $k < m$ fijos. El sistema de ecuaciones formado por relaciones del tipo:

$$d(P_i, P_j) = d_{ij} \pm e_{ij} \quad i, j = 1, \ldots, m$$

donde $d(P_i, P_j)$ representa la distancia euclidiana entre los puntos P_i y P_j, y e_{ij} representa la incertidumbre asociada a la medición de esta distancia; es un *sistema de ecuaciones de distancia con incertidumbre.*

Existen diferentes enfoques aplicados para resolver un sistema como el planteado anteriormente. Por ejemplo, en [Bac98] se analiza el problema de predecir la estructura molecular de una proteína, utilizando un lattice model (modelo basado en enrejados). Por su parte, en

[KB99], se aplica un modelo simplificado basado en cajas, que aproxima las esferas en torno a los puntos de referencia y calcula regiones factibles utilizando técnicas de programación con restricciones. Por su parte en [Tou02], se estudia la aplicación combinada de técnicas de consistencia interna y externa para el tratamiento de incertidumbres en CSP de distancia. La metodología propuesta en este último considera la aplicación de los siguientes pasos:

1. Solucionar el sistema de ecuaciones sin considerar la influencia de las incertidumbres. Es decir, encontrar todas las soluciones al problema, considerando la medición con precisión absoluta.

2. Agregar las incertidumbres al sistema, transformando las ecuaciones iniciales en inecuaciones que contemplen los límites (cotas máximas conocidas) para cada una de las mediciones. Por cada solución conocida, extender los dominios de las variables en forma consistente hasta obtener un intervalo de tolerancia en cada una.

3. Separar los conjuntos solución para cada una de las variables del problema y aplicar una técnica de consistencia externa para obtener el mínimo intervalo conteniendo el conjunto completo de solución.

1.4. Ámbito del Trabajo

1.4.1. Motivación

La motivación del trabajo es la de investigar de qué forma se generalizan las técnicas de Programación con Restricciones para ser aplicadas sobre dominios continuos (reales). En particular, estudiar la metodología para el tratamiento de incertidumbre en problemas de ecuaciones de distancia y de qué forma mejorar esta propuesta.

Del estudio de esta metodología se identificaron varios puntos susceptibles de mejorar. En particular referidos a:

1. Procedimientos de separación de puntos.

2. Estudio posterior de resultados obtenidos.

3. Análisis previo del problema.

El trabajo más profundo se centró en el primer punto, dado que constituye uno de los pilares fundamentales de la metodología. Se asume inicialmente que los puntos son separables, pero aun así la separación podría caer en cálculos erróneos en las cajas exteriores.

El segundo punto se centra en la imposibilidad de reducir a cero el error en el cálculo de cajas exteriores. Dado que el punto de corte separa el espacio de búsqueda inicial en subespacios disjuntos, una separación incorrecta produce cajas exteriores que coinciden con los límites del subespacio. Además, puede existir más de una caja exterior mal calculada, por lo que sería posible determinar qué puntos tuvieron problemas de separación.

Por último, para el tercer punto, se estudian propiedades del problema que puedan ser utilizadas para la implementación de un algoritmo de análisis previo, que permita reducir los errores de cálculo, previendo posibles problemas de separación.

1.4.2. Hipótesis

A partir de las observaciones previas se plantean las siguientes hipótesis:

1. La implementación de un nuevo algoritmo de separación de puntos incrementa el desempeño de la metodología, reduciendo los errores en el cálculo de cajas exteriores.

2. Un algoritmo de análisis posterior de resultados permite identificar errores previos de separación y facilita la correcta interpretación de resultados.

1.4.3. Organización de la Tesis

Esta Tesis está organizada de la siguiente manera:

En el capítulo 2, se presentan los Problemas de Satisfacción de Restricciones y sus definiciones asociadas. Además se examinan las nociones de complejidad asociada al problema y al algoritmo de búsqueda de solución. El capítulo 3 está dedicado al estado del arte de la Programación con Restricciones, particularmente sobre dominios continuos. En él, se examinan los conceptos relacionados con la aritmética de intervalos y las técnicas de consistencia aplicadas, tanto a dominios discretos como a dominios continuos. En el capítulo 4 se presenta una metodología de resolución de sistemas de ecuaciones de distancia con incertidumbre a través de un ejemplo concreto. La contribución de esta Tesis comienza en el capítulo 5, en donde se presenta un conjunto de observaciones relacionadas con los sistemas de ecuaciones

de distancia con incertidumbre y la influencia de estas observaciones en la metodología actual. En el capítulo 6 se presenta un nuevos algoritmo de separación de puntos, basado en las observaciones del capítulo anterior, que mejora las aproximaciones de la metodología actual y un algoritmo de análisis posterior que permite detectar errores en la separación de conjuntos solución. Posteriormente, en el capítulo 7 se presentan las conclusiones y propuestas para trabajos futuros.

Parte II

Estado del Arte

Capítulo 2

Problemas de Satisfacción de Restricciones

En este capítulo se examinan las principales definiciones relacionadas con los Problemas de Satisfacción de Restricciones. Además, se presentan ejemplos de CSP sobre dominios discretos y continuos. Finalmente se revisan las nociones de complejidad, asociadas tanto a las características del problema en sí, como al algoritmo de búsqueda de solución.

Los CSP tienen una gran importancia en diversas áreas del quehacer humano. Muchos de los problemas de la vida cotidiana pueden ser modelados a través de un conjunto de entidades denominadas **variables**, a las cuales se asocian conjuntos de valores posibles denominados **dominios** y que poseen conjuntos de relaciones entre sí, las cuales reciben el nombre de **restricciones**.

En este contexto un modelo es, entonces, una representación simplificada de la realidad a través de estas entidades.

2.1. Definiciones previas

Dada la gran cantidad de terminologías existentes para denotar un CSP, es necesario formalizar su definición y la de sus componentes relacionados. En este trabajo, se tratará un subconjunto de los problemas de satisfacción de restricciones, que corresponden a los CSP con dominios continuos que involucran ecuaciones de distancia. La definición formal de un CSP se presenta a continuación:

Definición 2.1.1 (CSP). Un problema de satisfacción de restricciones $P = (X, D, C)$ es una tripleta con $X = \{x_1, ..., x_n\}$ conjunto de variables, $D = \{D_{x_1}, ..., D_{x_n}\}$ conjunto de dominios asociados a las variables y $C = \{c_1, ..., c_m\}$ conjunto de restricciones sobre las variables.

Definición 2.1.2 (Espacio de búsqueda). Sea $P = (X, D, C)$ un problema de satisfacción de restricciones con $D = \{D_{x_1}, ..., D_{x_n}\}$. El espacio de búsqueda E asociado a P se define como $E = D_{x_1} \times ... \times D_{x_n}$ el producto cartesiano de los dominios asociados a cada una de las variables del problema.

Definición 2.1.3 (Solución). Una solución $\vec{s} = (a_1, ..., a_n)$ de un CSP $P = (X, D, C)$ es una asignación de valores a las variables donde $x_1 = a_1 \in D_{x_1}$, ... , $x_n = a_n \in D_{x_n}$, tal que $\forall\ c \in C, c(\vec{s})$ es verdadera.

Definición 2.1.4 (Espacio solución). Sea $P = (X, D, C)$ un problema de satisfacción de restricciones con espacio de búsqueda E. Sea $S \subset E$. Se dice que S es el espacio solución de P si $\forall\ \vec{s} \in S$, $\vec{s}$ es solución del problema y $\forall\ \vec{e} \in E - S$, $\vec{e}$ no es solución del problema.

En las definiciones previas no se ha restringido el tipo de dominios asignados a las variables ni las características de las restricciones. Si bien el dominio de las variables depende del problema en sí, es posible clasificarlo en dos grandes grupos: *dominios discretos* y *dominios continuos*. Las figuras 2.1 y 2.2 muestran dos ejemplos de CSP. El primero corresponde a un problema clásico de satisfacción de restricciones con dominios discretos denominado *coloreo de grafos*, mientras que el segundo corresponde a un problema con dominios continuos de ecuaciones de distancia.

1. Problema de *coloreo de grafos con 3 colores*: El problema de la figura 2.1 consiste en un grafo de tres nodos: X, Y, Z. Para cada nodo se debe seleccionar un color desde su dominio, de tal forma que dos nodos conectados por un arco no posean igual color. La solución al problema consiste en determinar una asignación simultánea de colores para todos los nodos del grafo, asegurando que todo par de nodos adyacentes posea diferente color. Por ejemplo, la asignación $Nodo(X) =$ **negro**, $Nodo(Y) =$ **azul** y $Nodo(Z) =$ **rojo**, es una solución al problema.

   Dado que el dominio de las variables es discreto, se dice que el CSP es discreto, y más específicamente, un *CSP sobre dominios discretos*. Una aplicación directa de este tipo

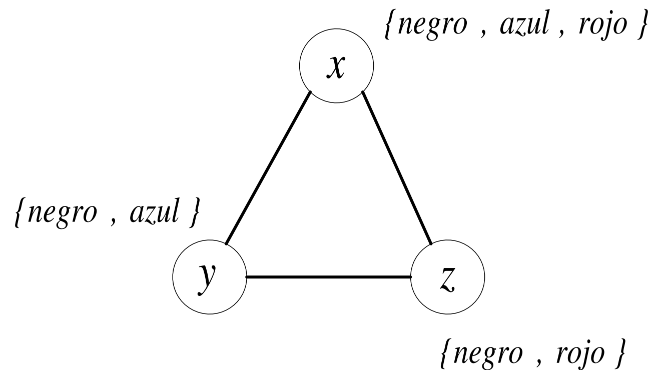


Figura 2.1: Ejemplo de un CSP sobre dominios discretos

de problemas se encuentra en el diseño de mapas. El objetivo es colorear un mapa respetando un conjunto máximo de colores, y asegurando que dos regiones colindantes del mapa no posean igual color. Formalmente el problema consiste en resolver $P = (X, D, C)$ con $X = \{x, y, z\}$, $D = \{\{negro, azul, rojo\}, \{negro, azul\}, \{negro, rojo\}\}$, y $C = \{\{x \neq y\}, \{x \neq z\}, \{y \neq z\}\}$.

La versatilidad de este problema permite usarlo y aplicarlo a problemas de asignación horaria y planificación, entre otros.

2. Problema de *ecuaciones de distancia*: El problema de la figura 2.2 consiste en dos puntos en el plano con ubicación fija X e Y, y un tercer punto Z que debe ser posicionado a una distancia no mayor a r_1 del punto X y r_2 del punto Y. Formalmente se trata de resolver $P = (X, D, C)$ con $X = \{\vec{x}, \vec{y}, \vec{z}\}$, $D = \{D_x = D_y = D_z = [0, 100] \times [0, 100]\}$, y $C = \{\{\vec{x} = \vec{a}\}, \{\vec{y} = \vec{b}\}, \{d(\vec{z}, \vec{x}) \leq r_1\}, \{d(\vec{z}, \vec{y}) \leq r_2\}\}$. La solución a este problema consiste en determinar la zona del plano que cumple con todas las restricciones simultáneamente.

   La zona demarcada representa todos aquellos puntos del plano que cumplen con las restricciones planteadas. Dado que el dominio de las variables (punto Z) es continuo, existen infinitas soluciones para el problema. Este tipo de problema recibe el nombre de *CSP numérico* ó *CSP sobre dominios continuos*. Una aplicación directa de este tipo de problemas se encuentra en la determinación de zonas de interferencia entre dos torres emisoras de ondas con cobertura limitada. En este caso interesa determinar las regiones del espacio que podrían presentar problemas de interferencia, debido a la influencia de

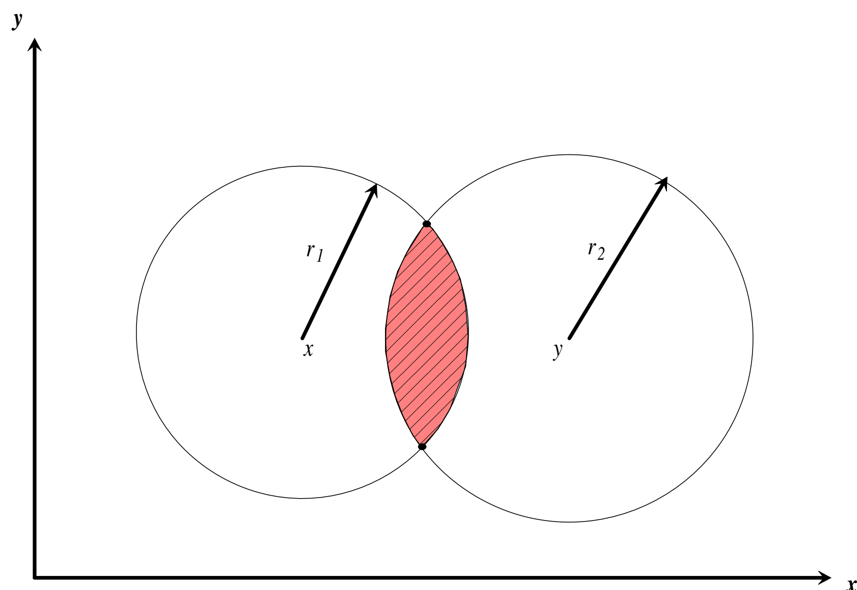


Figura 2.2: Ejemplo de un CSP sobre dominios continuos

diferentes ondas emanadas de artefactos emisores fijos.

Existe además una tercera clasificación de problemas: los CSP mixtos, que son aquellos que involucran variables con dominios discretos y continuos. Esta clase de CSP se denominan comúnmente *CSP híbridos*, y poseen un grado de complejidad adicional, ya que requieren una combinación de técnicas diseñadas para cada tipo de dominio. Un listado general de ejemplos de CSP y técnicas aplicadas para su resolución se pueden consultar en [Kum92].

2.2. Complejidad

El tema de la complejidad puede verse de dos formas distintas: *algoritmo de búsqueda de solución* y *problema en sí*. La principal diferencia entre ambas es el centro de estudio. Mientras la complejidad del algoritmo estudia las características propias de las técnicas para resolver un problema, la complejidad del problema estudia las características del problema en sí. Ambas alternativas poseen gran importancia en el estudio y solución de problemas complejos y pueden ser complementadas. La primera sirve para comparar dos algoritmos que resuelven el mismo problema y la segunda permite clasificar problemas basados en la dificultad de su resolución.

2.2.1. Complejidad algorítmica

Frecuentemente es necesario comparar algoritmos para medir cuál de ellos posee el mejor desempeño. Existen básicamente dos formas de realizar esta comparación: *la evaluación comparativa* y el *análisis de algoritmo*.

Evaluación comparativa

Consiste en ejecutar ambos programas en el mismo entorno (misma máquina) con un sistema operativo, un compilador y un conjunto de datos de entrada en particular y medir cual de ellos se ejecuta en forma más rápida o cual de ellos necesita menos memoria para su ejecución.

El principal problema de este tipo de comparación es la particularidad de la medición. Por esta razón, este método de comparación es poco utilizado en la práctica, y sólo está limitado a sistemas muy particulares en entornos muy estáticos.

Análisis de algoritmo

Este método es independiente de las entradas y del tipo de implantación realizada. Consiste básicamente en realizar un análisis de la cantidad de tareas (instrucciones) que realizará cada algoritmo abstrayendo las entradas y su implantación. El análisis se realiza en dos pasos:

1. **Abstrayendo la entrada**. Es decir, caracterizando al problema a través de un parámetro en particular. Una posible medida es la longitud de la secuencia (cantidad de variables o valores de entrada) que se caracteriza con un número n.

2. **Abstrayendo la implantación**. Es decir, encontrando una medida de desempeño que no dependa de la máquina, sistema operativo o compilador utilizado. Una posible medida es la cantidad de líneas de código ejecutadas, o la sumatoria de las instrucciones básicas realizadas (sumas, restas, asignaciones, comparaciones, entre otras). De esta forma se obtiene una caracterización, basada en los datos de entrada, denominada $T(n)$.

Si bien este enfoque representa mejoras con respecto al anterior, su utilización no está carente de problemas. El principal problema radica en que la ejecución de un programa no es independiente de los datos de entrada, por lo que el cálculo exacto del tiempo requerido $T(n)$

para una entrada de datos n, no es una medida de gran utilidad (ya que no posee una única respuesta).

Por otro lado, es posible determinar una función $T(n)$, suponiendo la utilización de diferentes tipos de entrada, y obtener un tiempo $T_{peor}(n)$, en el peor de los casos; $T_{mejor}(n)$, en el mejor de los casos; y un tiempo $T_{prom}(n)$ que denotaría el tiempo promedio de ejecución del algoritmo suponiendo una determinada distribución de datos de entrada.

De esta forma se obtiene la caracterización general de un algoritmo denominada *aproximación $O()$*.

Definición 2.2.1. Un algoritmo es $O(n)$ si su medida es a lo más una constante múltiplo de n, con la posible excepción de algunos casos particulares. De manera general, se tiene que:

$$T(n) \text{ es } O(f(n)) \iff \exists\, k, n_0 \in N \;/\; T(n) \leqslant kf(n)\ \forall\, n > n_0.$$

Esta aproximación es la más utilizada en el análisis de algoritmo, ya que representa un buen compromiso entre precisión y facilidad de análisis.

Es importante destacar que, si bien es difícil expresar la diferencia entre dos algoritmos con igual función $O()$ sin recurrir a un análisis más profundo, es relativamente claro que un algoritmo $O(n)$ poseerá mejor desempeño que uno $O(n^2)$, salvo en algunos casos particulares.

Otro punto importante, relacionado con los algoritmos utilizados para resolver CSP, es que la comparación se realiza usando como medida en número de chequeo de restricciones (y no el número de instrucciones de máquina, como se mostró en los casos previos).

Una explicación más detallada acerca de complejidad en análisis de algoritmos puede ser obtenida en [AVA88].

2.2.2. Complejidad de un problema

De igual forma que se realiza el análisis de algoritmos para determinar su complejidad, se estudian las características de los problemas que los hacen más o menos difíciles de resolver.

La principal diferencia global entre los problemas es el tiempo requerido para resolverlos. En primer lugar se encuentran aquellos que son susceptibles de ser resueltos en un tiempo polinomial. A ellos se les clasifica comúnmente como problemas P. Por otra parte, existen aquellos problemas para los cuales no existe aun un algoritmo capaz de resolverlos en este tiempo (para cualquier instancia particular del problema), por lo que se los clasifica como problemas NP.

Los problemas pertenecientes a la clasificación P son en general sencillos, ya que suelen ser resueltos por algoritmos del tipo $O(f(n))$ donde $f(n)$ representa un polinomio particular. A esto problemas se les denomina genéricamente *manejables*.

Por otra parte, los problemas clasificados como NP se caracterizan porque el tiempo necesario para resolverlos crece, al menos, exponencialmente ante un crecimiento lineal en el tamaño de la entrada. Esto dificulta su resolución, incluso para entradas de tamaño reducido.

Una subclasificación de problemas NP es la denominada NP *completos*. Este tipo de problemas corresponde a los más difíciles de resolver de la clase NP. Se ha demostrado que si es posible resolver uno de estos problemas en tiempo polinomial (para cualquier instancia particular), entonces todos ellos pueden ser resueltos en tiempo polinomial.

Una explicación más detallada acerca de complejidad e intratabilidad puede ser obtenida en [GJ79].

2.3. Resumen del capítulo

En este capítulo se han definido los conceptos relacionados con los Problemas de Satisfacción de Restricciones, tales como el espacio de búsqueda, solución y espacio solución. Se entregaron dos ejemplos de CSP, uno sobre dominios discretos y otro sobre dominios continuos. Además, se presentaron las nociones de complejidad asociada a un problema en sí y al algoritmo de búsqueda de solución.

Capítulo 3

Programación con Restricciones

La programación con restricciones involucra tanto el modelamiento como la resolución de sistemas basados en restricciones. Como disciplina abarca áreas del conocimiento y la investigación como las *matemática discretas*, el *análisis de intervalos*, la *programación matemática*, la *inteligencia artificial* y el *cálculo formal*.

Entre sus principales campos de aplicación se encuentran los *problemas de optimización combinatoria y ordenamiento*, *análisis financiero*, *diseño y construcción de circuitos integrados* [BBM94], *la biología molecular* [KB99] y *la resolución de problemas geométricos* [Cha93], entre otras.

Como idea general, la programación con restricciones intenta diseñar estrategias de resolución eficientes, usando el conocimiento y la estructura del problema.

En este capítulo se presenta una introducción de la Programación con Restricciones desde sus inicios, comenzando con los métodos de resolución de problemas restringidos a variables lineales, y continuando con sus extensiones en dominio y complejidad. Posteriormente se presenta un descripción detallada de la aritmética de intervalos, seguida de las técnicas de consistencia para problemas con dominios discretos y su extensión a problemas con dominios continuos. Finalmente se muestra un ejemplo que unifica los conceptos presentados en el capítulo.

3.1. Introducción

Uno de los primeros indicios de la programación con restricciones se encuentra en el año 1947, cuando George Dantzig crea el método simplex para programación lineal, descrito

por primera vez en su paper *Programming in a linear structure* [Dan48] (Programación en una estructura lineal). El término *programming* (programación) estaba referido a los tipos de problemas abordados por Dantzig en aquellos años, denominados *programming problems* (problemas de programación), relacionados con la investigación en *devise programs of activities for future conflicts* del departamento de defensa de los Estados Unidos. Posteriormente se utiliza el término *programación lineal* en lugar de *programación en una estructura lineal*, y los problemas pertenecientes a esta área reciben el nombre de *problemas de programación lineal*. De esta forma se asocia el término *programación* con un tipo de problema matemático específico en la literatura de la Investigación de Operaciones.

La programación con restricciones, también denominada *constraint logic programming* (Programación lógica con restricciones) comienza a desarrollarse a mediados de los años 80, como una técnica de la informática que combina el desarrollo de la comunidad de Inteligencia Artificial con los avances en lenguajes de programación. En este contexto, la palabra *programming* se refiere específicamente a un programa computacional. De esta forma, la programación con restricciones es en sí, una técnica de programación de computadores desde un punto de vista lógico. La programación lógica se caracteriza por ser declarativa y utilizar un estilo relacional basado en la lógica de primer orden.

Si bien en sus comienzos se utilizaban algoritmos simples para resolver las estructuras lógicas de un problema, el avance en investigación de nuevas técnicas y algoritmos más poderosos han extendido enormemente sus potencialidades.

Desde el punto de vista de la arquitectura, la programación con restricciones posee dos niveles [Hen99]: *el componente restringido* y *el componente de programación*. El primer nivel permite formular la definición del problema desde el punto de vista de sus variables y restricciones, de una forma relativamente simple y sin necesidad de grandes conocimientos de lenguajes de programación. Por otro lado, el componente de programación permite escribir algoritmos específicos para indicar de qué manera deberán ser modificadas las variables para encontrar valores que satisfagan las restricciones.

Para facilitar la aplicación de este tipo de técnicas a problemas complejos se han desarrollado diversos lenguajes de programación, que van desde **ALICE** [Lau78] en la década del 70 hasta los más actuales, como **OPL** [PVH99, Hen99] o **ILog** [ILO00b, ILO00a], a fines de la década del 90.

Una descripción más detallada de los inicios de la programación con restricciones y su evolución, puede ser obtenida en [LP01].

Dado el tema de esta Tesis se profundizará en los conceptos relacionados con intervalos. La siguiente sección presenta las definiciones vinculadas a la Aritmética de Intervalos.

3.2. Aritmética de Intervalos

Una amplia gama de problemas industriales requieren la resolución de CSP con restricciones sobre los reales. Como se vio en la sección 2.1, un *CSP numérico* involucra variables con dominios continuos (subconjuntos de los reales), que requieren de técnicas específicas de manipulación de intervalos para su resolución. En términos generales, la aritmética de intervalos puede ser descrita como *un método para tomar un conjunto de relaciones matemáticas entre ciertas cantidades, y construir a partir de ellas un proceso que mapea intervalos iniciales en intervalos finales para todas ellas* [OV93].

La figura 3.1 muestra un ejemplo de la aplicación de la aritmética de intervalo para reducir el dominio inicial de las variables pertenecientes a un problema particular.

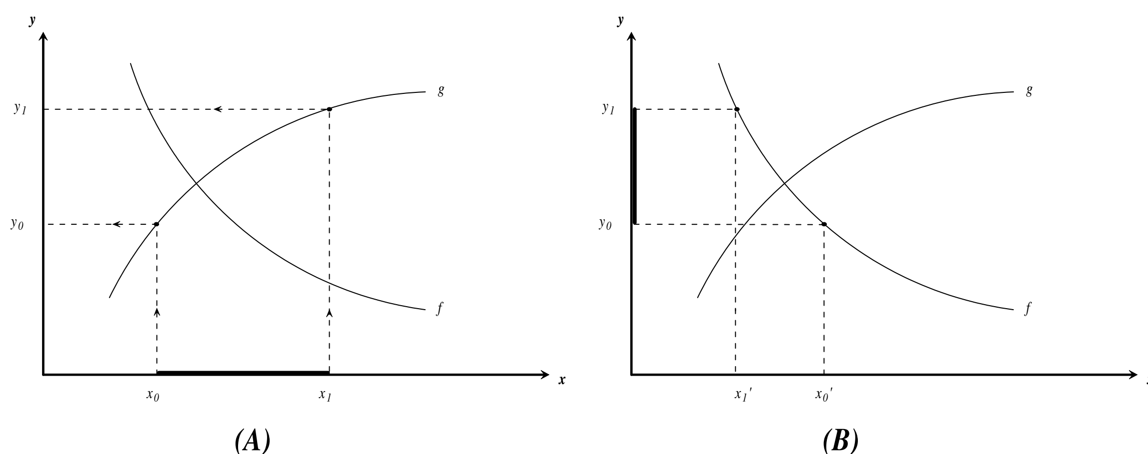


Figura 3.1: Ejemplo de aplicación de aritmética de intervalos.

El problema consiste en dos funciones $y = f(x)$ e $y = g(x)$ en el plano. Ambas poseen propiedades de monotonía deseables y un punto de intersección que interesa encontrar. A grandes rasgos, la estrategia consiste en partir de un intervalo dado $[x_0, x_1]$ por ejemplo, y proyectar los valores para una función determinada, sobre el eje contrario. En **(A)** se muestra la proyección $[y_0, y_1]$ obtenida como $y_0 = f(x_0)$ e $y_1 = f(x_1)$, que corresponde a la imagen del intervalo original. A partir de ella, se obtiene el intervalo de llegada, como preimagen

de la proyección obtenida y considerando la segunda función **(B)**. El nuevo intervalo $[x'_1, x'_0]$ (obtenido de $x'_0 = g^{-1}(y_0)$ y $x_1 = g^{-1}(y_1)$) es intersectado con el intervalo original para obtener el nuevo dominio.

Una observación importante del problema planteado, es que si:

- $g(x)$ es monótona creciente
- $f(x)$ es monótona decreciente
- La intersección del intervalo original y el final es vacía

se puede asegurar que el intervalo original no contenía al punto solución. Por el contrario, si el intervalo original contiene al punto solución, la intersección nunca será vacía.

A continuación se entregan un conjunto de definiciones que permiten explicar de una manera más formal los conceptos relacionados con la aritmética de intervalos.

3.2.1. Definiciones en aritmética de intervalos

Debido a las limitaciones que poseen los computadores para representar de manera exacta el conjunto de los números reales, es necesario distinguir el conjunto de valores *representables* del conjunto *real* [SSHF01, Rue02]. Para ello, el conjunto de los números reales $\mathbb{R}$ extendido con los dos símbolos de infinito y denotado por $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, +\infty\}$, es aproximado por el subconjunto $\mathbb{F}^\infty$, que usualmente corresponde a los números de punto flotante usados en la implementación de una máquina.

Por otro lado, si $a \in \mathbb{F}^\infty$, entonces a^+ corresponde al menor valor de $\mathbb{F}^\infty$ estrictamente mayor que a. Análogamente a^- corresponde al mayor valor de $\mathbb{F}^\infty$ estrictamente menor que a. Estos valores tienen sentido al considerar que la máquina posee precisión finita, en otra palabras, un conjunto limitado de valores.

Definición 3.2.1 (Intervalo). Un intervalo $[a, b]$ con $a, b \in \mathbb{F}$ es el conjunto de números reales $\{ r \in \mathbb{R} \mid a \leqslant r \leqslant b \}$

Adicionalmente, si r es un número real, es decir $r \in \mathbb{R}$, $\tilde{r}$ corresponde al más pequeño intervalo $[a, b]$ que contiene a r.

Definición 3.2.2 (Aproximación de conjunto). Sea S un subconjunto de $\mathbb{R}$, La aproximación de S, denotada por $\Box S$, es el menor intervalo I tal que $S \subseteq I$.

Esta definición corresponde a la extensión de conjuntos de la definición anterior. Si el conjunto S corresponde a un punto aislado p, entonces $\Box S = \tilde{p}$.

Definición 3.2.3 (k-box). Un k-box $B = I_1 \times \ldots \times I_k$ es la parte de un espacio k-dimensional definida por el producto cartesiano de los intervalos $(I_1, \ldots, I_k)$.

Definición 3.2.4 (Extensión de los intervalos). Una función definida sobre un conjunto de intervalos $F : I^n \mapsto I$, es una extensión sobre los intervalos de $f : R^n \mapsto R$ si y sólo si $\forall\ I_1, \ldots, I_n \in I : r_1 \in I_1, \ldots, r_n \in I_n \Rightarrow f(r_1, \ldots, r_n) \in F(I_1, \ldots, I_n)$.

De igual forma, una relación sobre un conjunto de intervalos $C : I^n \mapsto Bool$, es una extensión sobre los intervalos de la relación definida sobre los reales $c : R^n \mapsto Bool$ si y sólo si $\forall\ I_1, \ldots, I_n \in I : r_1 \in I_1, \ldots, r_n \in I_n \Rightarrow [c(r_1, \ldots, r_n) \Rightarrow C(I_1, \ldots, I_n)]$.

Por ejemplo, la relación sobre los intervalos $\doteq$ definida como $I_1 \doteq I_2 \Leftrightarrow (I_1 \cap I_2) \neq \emptyset$ es una extensión de los intervalos de la relación de igualdad sobre los reales.

Definición 3.2.5 (Extensión natural de los intervalos). Una función definida sobre un conjunto de intervalos $F : I^n \mapsto I$, es una extensión natural sobre los intervalos de $f : R^n \mapsto R$ si y sólo si F es una extensión sobre los intervalos de f obtenida al reemplazar en f cada constante k por su extensión natural sobre los intervalos $\tilde{k}$, cada variable por una variable sobre los intervalos y cada operación aritmética por su extensión óptima [Moo66] de los intervalos. De manera informal se define la *extensión óptima* como el intervalo más pequeño que conserva el conjunto de soluciones de una función continua.

Como ejemplo, se muestra la definición de la extensión óptima de los intervalos para las cuatro operaciones básica {+, -, ∗, /}:

- $[a, b] \oplus [c, d] = [a + c, b + d]$
- $[a, b] \ominus [c, d] = [a - d, b - c]$
- $[a, b] \otimes [c, d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]$
- $[a, b] \oslash [c, d] = [min(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}), max(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d})]$, siempre que $0 \notin [c, d]$

3.2.2. Propiedades

Con respecto a las propiedades, una observación importante es que la forma sintáctica de la función influye en la precisión de la extensión natural de los intervalos que le será asociada. Por ejemplo, las funciones $f_1(x) = x^2 - x$ y $f_2(x) = x(x-1)$ tienen (para un dominio dado $x = [0, 5]$) diferente valor de extensión natural $F_1([0, 5]) = [-5, 25]$ y $F_2([0, 5]) = [-5, 20]$. Por esta razón, si bien los operadores $\{+, *\}$ cumplen con las propiedades de conmutatividad y asociatividad, no cumplen con la propiedad distributiva. Otras propiedades interesantes son las siguientes:

Proposición 3.2.1. *Sea $F : I^n \mapsto I$ la extensión natural de los intervalos de $f : R^n \mapsto R$ y sea $f_{sol} = \Box\{f(r_1, \ldots, r_n) \mid r_1 \in I_1, \ldots, r_n \in I_n\}$. Si cada r_i posee sólo una ocurrencia en f, entonces $f_{sol} = F(I_1, \ldots, I_n)$. En caso contrario $f_{sol} \subseteq F(I_1, \ldots, I_n)$.*

Proposición 3.2.2. *Sea $C : I^n \mapsto Bool$, la extensión natural de los intervalos de una ecuación $c : R^n \mapsto Bool$. Si cada x_i posee sólo una ocurrencia en la ecuación c, entonces $C(I_1, \ldots, I_n) \Leftrightarrow (\exists\ x_1 \in D_{x_1}, \ldots, x_n \in D_{x_n} \mid c(x_1, \ldots, x_n)$ se cumple $)$.*

En la siguiente sección se describe un conjunto de técnicas de consistencia, partiendo por la arco-consistencia utilizada en la resolución de problemas con dominios discretos, para continuar con extensiones de esta técnica sobre dominios continuos.

3.3. Técnicas de Consistencia

Las técnicas de consistencia juegan un papel fundamental en la aplicación de algoritmos de propagación para reducir los dominios de las variables pertenecientes a un problema. De manera general, las técnicas de consistencia examinan el dominio de las variables en busca de valores o conjunto de valores que no podrían ser asignados respetando las restricciones del problema. Se denominan técnicas de filtrado y se aplican para encontrar un CSP equivalente[1].

Una de las técnicas de consistencia más difundida y utilizada para reducir dominios en CSP con dominios discretos en la arco-consistencia [Mac77]. Las técnicas discutidas en este trabajo, aplicadas sobre dominios continuos se han basado en arco-consistencia, relajando sus exigencias para ser aplicada sobre intervalos [Fal94].

[1]Se dice que un CSP P es *equivalente* a un CSP P' si ambos poseen el mismo espacio solución.

3.3.1. Consistencia en dominios finitos

arco-consistencia

El algoritmo para la arco-consistencia reduce los dominios de las variables eliminando aquellos valores que son inconsistentes (para una restricción dada) con los valores permitidos para las otras variables involucradas en la restricción. De manera formal, la arco-consistencia se define como:

Definición 3.3.1 (restricción arco-consistente). Sea $c(x_1, \ldots, x_n)$ una relación entre las variables $x_1, \ldots, x_n$ de dominios $D_{x_1}, \ldots, D_{x_n}$ discretos. La relación c es arco-consistente ssi $\forall\, v_i \in D_{x_i}, \exists\, (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \in D_{x_1} \times \ldots D_{x_{i-1}} \times D_{x_{i+1}} \ldots \times D_{x_n}$ tal que $c(v_1, \ldots, v_n)$ se cumple, $\forall\ i = 1, ..., n$.

Este tipo de consistencia es denominada *local*, ya que sólo examina la influencia de una restricción, y no el conjunto de restricciones del problema. A partir de ella se define un *CSP arco-consistente* como aquel que posee todas sus restricciones arco-consistentes.

Definición 3.3.2 (CSP arco-consistente). Sea $P = (X, D, C)$ un CSP. Se dice que P es *arco-consistente* ssi $\forall v_i \in D_{x_i}$, $\forall c \in C\ \exists (v_1, ..., v_{i-1}, v_{i+1}, ..., v_n) \in D_{x_1} \times ...D_{x_{i-1}} \times D_{x_{i+1}} ... \times D_{x_n}$ tal que $c(v_1, ..., v_n)$ se cumple, $\forall\ i = 1, ..., n$.

Es importante notar que el operador $\exists$ está ligado a cada restricción c, ya que debe *existir* una n-upla que cumpla la restricción (no necesariamente igual para todas las restricciones). La figura 3.2 muestra un ejemplo de CSP no arco-consistente **(A)** y un CSP arco-consistente **(B)**.

El ejemplo corresponde a un problema de coloreo de grafo con tres nodos. Cada nodo está conectado con los otros dos, por lo que no pueden compartir un mismo color. En **(A)**, la variable x_1 posee en color *negro* como parte del su dominio. Dado que la variable x_3 sólo posee el color *negro* en su dominio y la restricción $x_1 \neq x_3$ debe ser satisfecha, entonces $x_1 = negro$ no es una asignación consistente para el problema. En conclusión, el valor *negro* debe ser eliminado del dominio de x_1. Este análisis es independiente de las demás restricciones del problema, y puede ser realizado antes de comenzar una etapa de búsqueda de solución. Por otro lado, en **(B)** se tiene un CSP arco-consistente, ya que para cada valor perteneciente al dominio de una variable existe un valor en el dominio de las demás que satisface cada una de las restricciones.

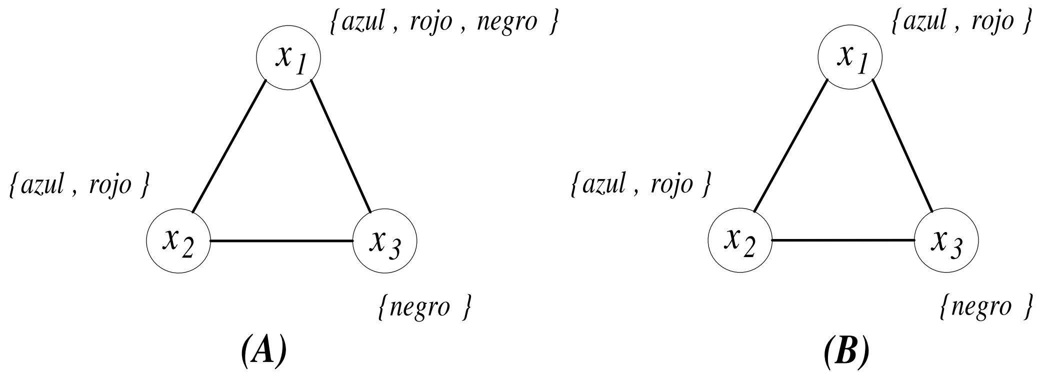


Figura 3.2: Ejemplo de un CSP no arco-consistente (A) y uno arco-consistente (B)

Es importante remarcar que un CSP arco-consistente no es sinónimo de un problema con solución. Las técnicas de consistencia sólo aseguran la incapacidad que tienen algunos valores pertenecientes a los dominios para satisfacer las restricciones del problema, pero nada dice acerca de si el problema tiene solución. La figura 3.3 muestra un ejemplo de CSP arco-consistente que no posee solución.

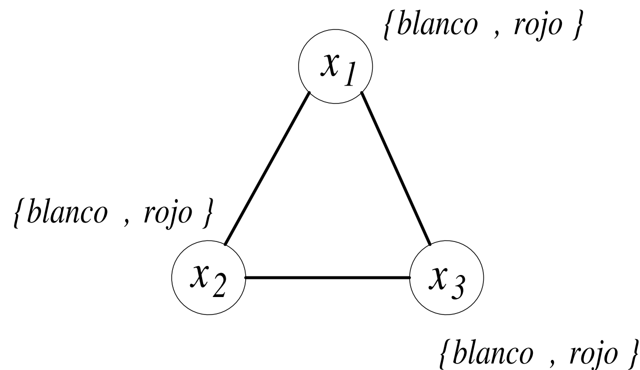


Figura 3.3: Ejemplo de un CSP arco-consistente sin solución

En el ejemplo, cada variable posee dos valores que son arco-consistente para cada una de las restricciones, pero la asignación de un valor para cualquiera de ellas produce un sistema inconsistente. No existe solución factible que respete simultáneamente todas las restricciones.

Definición 3.3.3 (Filtraje por arco-consistencia). Un filtraje por arco-consistencia de un CSP $P = (X, D, C)$ con $D = \{D_{x_1}, \ldots, D_{x_n}\}$ es un CSP $P' = (X, D', C)$ con $D' = \{D'_{x_1}, \ldots, D'_{x_n}\}$ tal que:

- P y P' tienen el mismo espacio solución.
- P' es arco-consistente.
- $D'_{x_i} \subset D_{x_i}$ $\forall$ i, y los dominios de P' son los más grandes de tal forma que P' sea arco-consistente.

La definición anterior no especifica el algoritmo utilizado para filtrar los dominios de las variables hasta conseguir un CSP arco-consistente. En la literatura existen diferentes algoritmos que realizan este procedimiento, y su comparación se basa en la eficiencia (complejidad algorítmica). Entre ellos se pueden destacar **AC-1**, **AC-2**, **AC-3**, **AC-4** y **MAC** [Mac77, Dec01, Mil03] como los más difundidos.

3.3.2. Consistencia externa en dominios continuos

En esta sección se estudia más detalladamente las técnicas de consistencia utilizadas para filtrar dominios de CSP numéricos. Dado que existen diferentes técnicas para aproximar el conjunto solución de un CSP, se han integrado conceptualmente bajo el nombre de *e-consistencia* o *consistencia externa*. En lo sucesivo, se entenderá por consistencia externa a una de las técnicas de consistencia aquí presentadas (la que produzca la mejor aproximación). De manera general, un filtraje por e-consistencia de un CSP, será aquel que calcula la más pequeña caja exterior (véase definición 3.2.3) que contiene a su conjunto solución.

Adicionalmente se utilizará como notación: $\Phi_{tecnica}(P)$ donde *técnica* = {*2B, 3B, Box, Bound* }, para denotar un CSP P filtrado por la técnica respectiva. Además, la notación $\Phi_A(P) \preceq \Phi_B(P)$ se utilizará para indicar que el método A produce una reducción al menos igual que el método B. Por último, la notación $\Phi_A(P) \equiv \Phi_B(P)$ se usará para indicar que ambas técnicas producen reducciones similares.

En las siguientes secciones se hará una revisión de las 4 técnicas de reducción más utilizadas en el manejo de intervalos.

2B-consistencia

Conceptualmente la 2B-consistencia es una aproximación de la arco-consistencia para dominios continuos [OV90]. Una restricción c es 2B-consistente si, para toda variable x con dominio $D_x = [a, b]$ existen valores en el dominio de las otras variables que satisfacen la restricción c cuando x es instanciada con a y cuando x es instanciada con b. En otras

palabras, si c puede ser escrita de la forma $f(x) = 0$, entonces a y b corresponden al menor y mayor cero de f en el espacio delimitado por $D_{x_1} \times \ldots \times D_{x_n}$. Formalmente la 2B-consistencia se define como:

Definición 3.3.4 (2B-consistencia). Sea $P = (X, D, C)$ un CSP con $D = \{D_{x_1}, \ldots, D_{x_n}\}$. La restricción k-aria $c \in C$ es 2B-consistente ssi: $\forall i$, $D_{x_i} = \Box\{v_i \in D_{x_i}$ tal que $\exists v_1 \in D_{x_1}, \ldots, \exists v_{i-1} \in D_{x_{i-1}}, \exists v_{i+1} \in D_{x_{i+1}}, \ldots, \exists v_n \in D_{x_k}$ tal que $c(v_1, \ldots, v_{i-1}, x_i, v_{i+1}, \ldots, v_k)$ se cumple $\}$. Un CSP se dirá 2B-consistente si todas sus restricciones lo son.

A diferencia de la noción de arco-consistencia, la 2B-consistencia no impone una restricción sobre todos los valores pertenecientes al dominio de la variable, sino sólo sobre los bordes (límite inferior y superior) del intervalo.

Definición 3.3.5 (Filtraje por 2B-consistencia). Un filtraje por 2B-consistencia de un CSP $P = (X, D, C)$ con $D = \{D_{x_1}, \ldots, D_{x_n}\}$ es un CSP $P' = (X, D', C)$, con $D' = \{D'_{x_1}, \ldots, D'_{x_n}\}$ tal que:

- P y P' tienen el mismo espacio solución.
- P' es 2B-consistente.
- $\forall i$, $D'_{x_i} \subset D_{x_i}$ y los dominios de P' son los más grandes de tal forma que P' sea 2B-consistente.

Una descripción más detallada de la 2B- consistencia y los algoritmos de filtraje puede ser obtenida en [Rue02].

3B-consistencia

El principio fundamental de la 3B-consistencia es comprobar que el sistema de restricciones no se convierta en trivialmente inconsistente [Lho93]. Esto debido a que la 2B-consistencia es aplicada considerando sólo una restricción a la vez. La 3B-consistencia examina los bordes del intervalo con el fin de comprobar que éste pueda efectivamente ser solución del problema.

Definición 3.3.6 (3B-consistencia). Sea $P = (X, D, C)$ un CSP y x una variable de X con dominio $[a, b]$. Sean $P_{D_x^1 \leftarrow [a,a^+)}$ el CSP derivado de P al sustituir el dominio D_x por $D_x^1 = [a, a^+)$; y $P_{D_x^2 \leftarrow (b^-,b]}$ el CSP derivado de P al sustituir el dominio D_x por $D_x^2 = (b^-, b]$. El dominio D_x es 3B-consistente si y sólo si $\Phi_{2B}(P_{D_x^1}) \neq P_\phi$ y $\Phi_{2B}(P_{D_x^2}) \neq P_\phi$. Un CSP será 3B-consistente si todos sus dominios lo son.

Box-consistencia

La Box-consistencia ha sido definida [BMV94] para remediar alguno problemas introducidos por la descomposición de restricciones en *primitivas* para el cálculo de la 2B-consistencia. A diferencia de ésta última, la Box-consistencia no requiere descomposición del sistema de restricciones para ser aplicada. La definición formal es:

Definición 3.3.7 (box-consistencia). Sea (X, D, C) un CSP y $c \in C$ una restricción k-aria definida sobre $(x_1, \ldots, x_k)$. Se dice que c es *Box-consistente* si: $\forall\ x_i \in \{x_1, \ldots, x_k\} \mid D_{x_i} = [a, b]$, las siguientes relaciones se cumplen:

- $C(D_{x_1}, \ldots, D_{x_{i-1}}, [a, a^+), D_{x_{i+1}}, \ldots, D_{x_k})$,
- $C(D_{x_1}, \ldots, D_{x_{i-1}}, (b^-, b], D_{x_{i+1}}, \ldots, D_{x_k})$

El filtraje por Box-consistencia se define de manera similar al filtraje por 2B-consistencia. Una observación importante es que $\Phi_{2B}(P) \preceq \Phi_{box}(P)$ y $\Phi_{2B}(P) \equiv \Phi_{box}(P)$ si las variables aparecen una sola vez en cada restricción.

Bound-consistencia

La Bound-consistencia tiene sobre la Box-consistencia el mismo efecto que la 3B-consistencia sobre la 2B-consistencia, es decir, examina los bordes del intervalo para evitar que el sistema se vuelva trivialmente inconsistente [ZW98]. En otras palabras, verifica si la box-consistencia puede ser aplicada cuando el dominio de una variable es reducido al valor de uno de sus bordes. Formalmente:

Definición 3.3.8 (bound-consistencia). Sea (X, D, C) un CSP P, y $c \in C$ una restricción de orden k. Se dice que c es *Box-consistente* si: $\forall\ x_i \in \{x_1, ..., x_k\} \mid D_{x_i} = [a, b]$, se cumple que:

- $\Phi_{box}(C(D_{x_1}, \ldots, D_{x_{i-1}}, [a, a^+), D_{x_{i+1}}, \ldots, D_{x_k})) \neq P_\emptyset$,
- $\Phi_{box}(C(D_{x_1}, \ldots, D_{x_{i-1}}, (b^-, b], D_{x_{i+1}}, \ldots, D_{x_k})) \neq P_\emptyset$

Se puede comprobar que, así como $\Phi_{box}(P) \preceq \Phi_{2B}(P)$, se tiene que $\Phi_{bound}(P) \preceq \Phi_{3B}(P)$

Cálculo de consistencia externa con análisis de intervalos

Finalmente se entrega un ejemplo del cálculo de consistencia externa utilizando la aritmética de intervalos [COP01]. El procedimiento consiste en determinar los límites del intervalo $[\underline{L}, \overline{R}]$ de tal forma que encierren todas las soluciones al problema, y no sea posible reducirlo. Para calcular el límite superior $\overline{R}$, el intervalo inicial $[a, b]$ es dividido en dos subespacios disjuntos $[a, \frac{a+b}{2}]$ y $[\frac{a+b}{2}, b]$, analizando la presencia de solución en el subespacio superior. Si éste no posee solución, se elimina. Por el contrario, si aun posee solución, este nuevo subespacio es dividido en dos, y se analiza el nuevo subespacio en forma recursiva. El proceso se detiene cuando el largo del intervalo es menor que una precisión establecida.

Por ejemplo, considérese la función $f(x) = x^2 - 2x + 1 = 0$, para un x en el intervalo $[2, 4]$. La determinación de R se realiza considerando inicialmente el intervalo $[3, 4]$. Como $f([3, 4]) = [3, 4]^2 - 2 * [3, 4] + 1 = [2, 11]$, vale decir, no contiene ceros para la función, el intervalo $[3, 4]$ es eliminado. De igual forma, para el cálculo de $[\underline{L}$, se analiza el intervalo $[2, 3]$, obteniéndose $f([2, 3]) = [-1, 6]$. Vale decir, no es posible descartar el intervalo, por lo tanto se procede al análisis del intervalo $[2, 2{,}5]$. La determinación de $[\underline{L}$ continúa con la siguiente progresión:

$$\begin{aligned} f([2, 2{,}5]) &= [0, 3{,}25] &\Rightarrow\ & L = [2, 3] \\ f([2, 2{,}25]) &= [0{,}5, 2{,}0625] &\Rightarrow\ & L = [2{,}25, 3] \\ f([2{,}25, 3]) &= [0{,}0625, 5{,}5] &\Rightarrow\ & x = \emptyset \end{aligned}$$

De esta forma, se determina que el intervalo $[2, 4]$ no posee ceros para el función $f(x) = x^2 - 2x + 1$.

3.3.3. Consistencia interna en dominios continuos

A diferencia de las técnicas de e-consistencia, que están basadas en la arco-consistencia para reducir los dominios de las variables, la i-consistencia [Hél99] es un *concepto* para denotar aquellos CSP compuestos por variables que sólo poseen dominios factibles. Formalmente se define como:

Definición 3.3.9 (i-consistencia). Sea $P = (X, D, C)$ un CSP con $D = \{D_{x_1}, \ldots, D_{x_n}\}$. P será i-consistente ssi $\forall c \in C$, $\forall (x_1, \ldots, x_n) \in D_{x_1} \times \ldots \times D_{x_n}$, $c(x_1, \ldots, x_n)$ se cumple.

En estricto rigor la definición implica que toda asignación de valores a las variables cumple todas y cada una de las restricciones del problema. Como ejemplo, suponga un problema

$P = \{X = \{x_1, x_2\}, D = \{[2,4], [5,8]\}, C = \{x_1 < x_2\}\}$. Se dice que P es i-consistente, ya que sólo posee asignaciones válidas en el dominio de sus variables.

Extensión de dominio i-consistente

La noción de i-consistencia permite desarrollar una metodología de extensión de dominio, que posee la característica de mantener sólo asignaciones válidas para cada variable. Formalmente:

Definición 3.3.10 (Extensión derecha i-consistente). Sea $P = (X, D, C)$ un CSP i-consistente. $P' = (X, D', C)$ es una extensión derecha i-consistente de D_x para P ssi:

- $\forall\ D_i \in D \backslash \{D_x\}, D_i = D'_i$
- $D_x \subset D'_x, \underline{D'_x} = \underline{D_x}$, es decir, conservan el mismo límite inferior.
- P' es i-consistente.

Análogamente se define una extensión izquierda i-consistente, teniendo en cuenta que los límites superiores de ambos dominios permanecen fijos.

Definición 3.3.11 (Función extrema). Sea c una inecuación de tipo $f(x_1, \ldots, x_n) \leqslant 0$ ó $f(x_1, \ldots, x_n) \geqslant 0$, c_{equ} denota la ecuación correspondiente a c, es decir $f(x_1, \ldots, x_n) = 0$. $F_c^{min(x)}(D)$ es una función extrema optimal de c_{equ} para la variable x si $F_c^{min(x)}(D)$ calcula el menor valor de x, el cual es solución de c_{equ} en el espacio delimitado por D.

Finalmente, la metodología propuesta en [Hél99] para la extensión derecha de dominios i-consistentes consta de los siguientes pasos:

1. Buscar un subconjunto del espacio solución del problema. Éste puede ser incluso un punto solución.
2. Seleccionar la variable a la cual se desea extender el dominio.
3. Definir, para cada inecuación, una función extrema que calcule la solución más a la izquierda de la variable seleccionada en el espacio definido por el dominio de las otras variables.
4. Encontrar la menor solución de todas las funciones extremas.

Ejemplo

Para ejemplificar la extensión de dominios consistentes en un CSP, considérese el CSP $P = (X = \{x_1, x_2\}, D = \{[0, 100], [0, 100]\}, C = \{x_1 * x_2 \leqslant 5\})$ mostrado en la figura 3.4. Suponga que se tiene un dominio inicial, que se sabe es solución del problema (dominio i-consistente), $D_{x_1} = [0, 1]$ y $D_{x_2} = [1, 2]$. Para realizar una extensión derecha del dominio D_{x_1} se considera la ecuación $x_1 * x_2 = 5$ en el espacio delimitado por $x_1 \in [1, 3]$ y $x_2 \in [1, 2]$. La solución más a la izquierda del nuevo problema es $x_1 = 2{,}5$ y $x_2 = 2$. Por lo tanto, el nuevo dominio i-consistente extendido por la derecha de x_1 es $D_{x_1} = [0; 2{,}5]$.

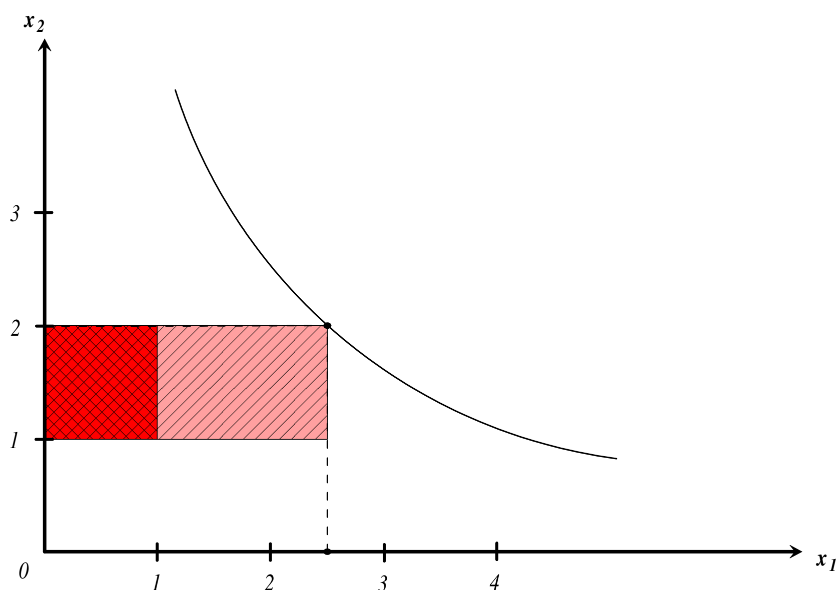


Figura 3.4: Extensión de dominios consistentes en CSP

3.4. Ejemplo gráfico de técnicas de consistencia

La figura 3.5 muestra la relación existente entre las técnicas de consistencia presentadas en las secciones 3.3.2 y 3.3.3, al ser aplicadas a un CSP.

Las zonas blancas representan el espacio solución de un CSP particular sobre el plano cartesiano. Se ha considerado como dominio inicial el primer cuadrante del plano. El rectángulo punteado externo, representa la aplicación de una técnicas de consistencia externa para aproximar las soluciones del CSP. Nótese que la aplicación de e-consistencia encierra todas las

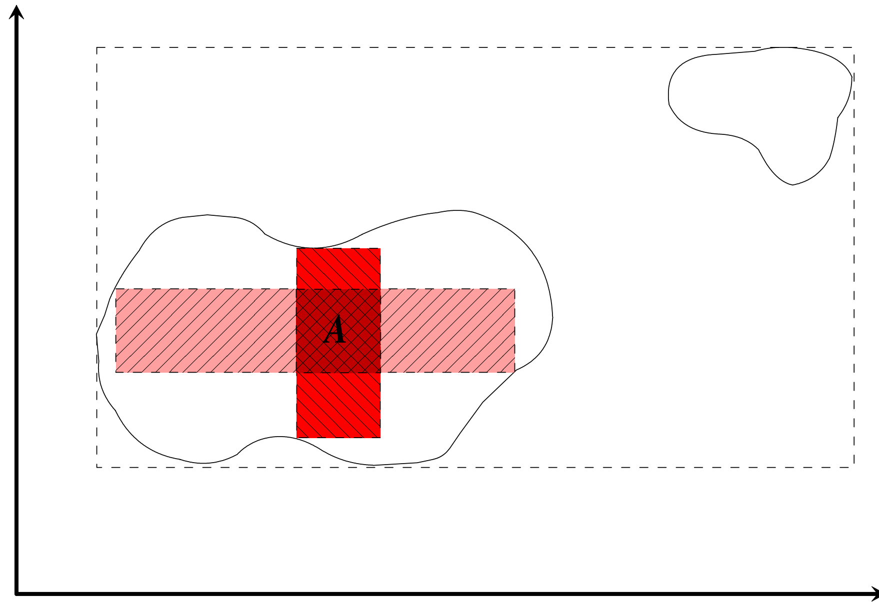


Figura 3.5: Relación entre la e-consistencia y la i-consistencia

soluciones del problema, abarcando además espacio que no es solución. El cuadrado pequeño, etiquetado como conjunto A, representa un conjunto solución inicial dado u obtenido con algún método de búsqueda de solución. Si al CSP original, se le restringe el dominio al espacio abarcado por el conjunto A, se tiene un CSP i-consistente. Por último, suponiendo que se tiene un CSP i-consistente y se desea extender el dominio de una variable, se obtiene como resultado el rectángulo horizontal inscrito en la zona blanca mayor (para una extensión sobre el eje X), y el rectángulo vertical inscrito en la zona blanca mayor (para una extensión sobre el eje Y) del dominio i-consistente.

3.5. Resumen del capítulo

En este capítulo se han definido los conceptos relacionados con la Programación con Restricciones, centrándose principalmente en aquellos relacionados con dominios continuos. Se presentaron las nociones básicas de la aritmética de intervalos, así como las técnicas de consistencia aplicadas a los CSP con dominios continuos. Se explicaron las técnicas de 2B-consistencia, 3B-consistencia, Box-consistencia y Bound-consistencia, agrupándolas bajo el concepto de *e-consistencia*. Además, se presentó una clase particular de consistencia, denominada *i-consistencia*, que consiste en una noción más que en una técnica de filtraje.

Por último, se presentó una técnica de extensión de dominios consistentes, que utiliza la i-consistencia para asegurar que el dominio final posea sólo soluciones al problema, y se mostró un ejemplo del resultado esperado por cada una de las técnicas.

Capítulo 4

Metodología de Resolución

Hasta ahora se han revisado los conceptos relacionados con los CSP y las técnicas de CP aplicadas a la resolución de esta clase de problemas. En lo que sigue, se presenta la metodología realizada por Julien Touati en el INRIA de Sophia-Antipolis, Francia. El objetivo de esta metodología es aproximar las soluciones de un sistema de ecuaciones de distancia, cuando la medida de la distancia posee un cierto grado de incertidumbre[1].

El trabajo completo puede ser obtenido en [Tou02]. El objetivo principal es el tratamiento de incertidumbre en restricciones de distancia y puede ser resumida en los siguientes pasos:

1. Encontrar todas las soluciones del sistema de ecuaciones sin considerar la influencia de las incertidumbres.

2. Incluir las incertidumbres del problema, transformado las ecuaciones en sus inecuaciones correspondientes y, para cada solución encontrada, extender los dominios de las variables en forma consistente para obtener una *caja interior* (mínimo espacio de tolerancia) para cada uno de los puntos pertenecientes a la solución.

3. Separar cada punto perteneciente a una solución, de sus alternativas existentes en otras soluciones, aislando su posible influencia en el cálculo posterior de *cajas exteriores*.

4. Por cada punto perteneciente a una solución, aplicar técnicas de consistencia externa para obtener una *caja exterior* (máximo espacio de tolerancia).

[1]La noción de incertidumbre corresponde al grado de tolerancia en la medida de la distancia, debido por ejemplo a la precisión del instrumento de medición, y que convierte a un *punto solución* en una *zona de factibilidad*. Esta incertidumbre se asume acotada y pequeña en comparación a la medida en sí.

En conclusión, se desea obtener una aproximación de las soluciones del sistema considerando la influencia que las incertidumbres provocan.

4.1. Explicación mediante un ejemplo

Antes de estudiar en detalle la metodología de Touati, se plantea un ejemplo simplificado para visualizar los diferentes pasos y sus influencias en la resolución de los sistemas planteados.

La figura 4.1 muestra un problema que consiste en encontrar la posición de un artefacto emisor de ondas que ha caído en una zona cercana a dos antenas de recepción. Dadas las características de la onda emitida, cada antena posee cierta información de la posible distancia a la que se encuentra el objeto. Por la magnitud de la señal, la primera antena determina que el objeto se encuentra a una distancia d_1 de ésta, mientras que la segunda antena determina que entre ella y el objeto existe una distancia d_2. Debido a las características de los aparatos de medición, no se asegura que la distancia calculada sea exacta, pero si es posible afirmar que el valor real no dista por más de i metros del valor propuesto para ambas distancias.

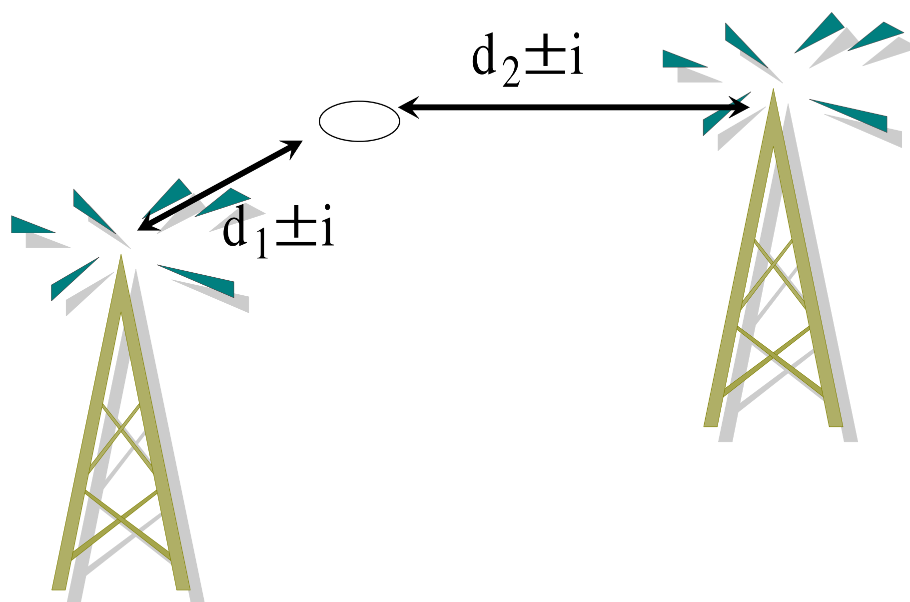


Figura 4.1: Ejemplo de un problema de ecuaciones de distancia

Dado que la ubicación en ambas antenas receptoras es bien conocida, y suponiendo ciertas características del terreno (plano), es posible simplificar el problema planteándolo de la

siguiente manera: *dado dos puntos P_1 y P_2 (fijos) en el plano cartesiano, se desea determinar la ubicación de un tercer punto P_3 que se encuentra a una distancia $d(P_1, P_3) = d_1 \pm i$ y $d(P_2, P_3) = d_1 \pm i$ a los primeros, respectivamente.*

En el problema se asume que la incertidumbre en ambas mediciones es la misma, y corresponde al valor de i.

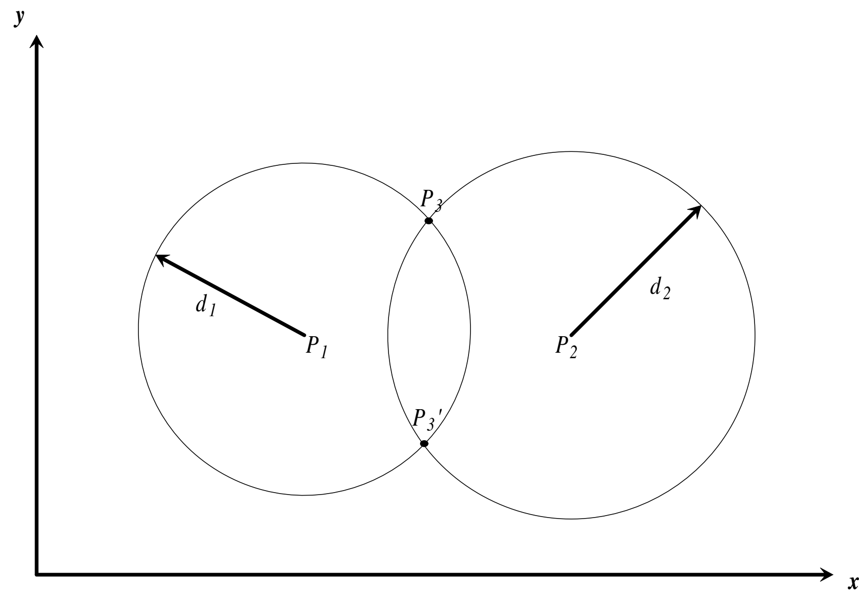


Figura 4.2: Simplificación del problema en un plano cartesiano

La figura 4.2 muestra el problema simplificado en un plano cartesiano, omitiendo la influencia de la incertidumbre en la medición de la distancia. Suponga que el punto P_1 tiene coordenadas $(p_{1a}, p_{1b}) = (0, 0)$ y el punto P_2 tiene coordenadas $(p_{2a}, p_{2b}) = (0, 5)$. Entonces, se desean determinar las coordenadas del punto $P_3 = (p_{3a}, p_{3b})$ que cumplen con las restricciones del problema. El sistema de ecuaciones que describe la situación planteada anteriormente puede ser escrito de la siguiente manera:

$$
\begin{aligned}
&p_{1a} = 0 \text{ y } p_{1b} = 0 && \text{(punto } P_1 \text{ fijo)}\\
&p_{2a} = 0 \text{ y } p_{2b} = 5 && \text{(punto } P_2 \text{ fijo)}\\
&d(P_1, P_3) = \sqrt{(p_{1a} - p_{3a})^2 + (p_{1b} - p_{3b})^2} = d_1 && \text{(distancia entre } P_1 \text{ y } P_3)\\
&d(P_2, P_3) = \sqrt{(p_{2a} - p_{3a})^2 + (p_{2b} - p_{3b})^2} = d_2 && \text{(distancia entre } P_2 \text{ y } P_3)
\end{aligned}
$$

Suponiendo que la situación sea, cualitativamente, la mostrada en la figura 4.2, la resolución del sistema de ecuaciones anterior entregaría dos posibles soluciones: los puntos P_3 y P_3' mostrados en ella.

Si bien es claro que estos puntos son solución del problema simplificado (al omitir la influencia de las incertidumbres), los valores obtenidos pueden no reflejar la posición exacta del objeto. Más aún, la solución exacta del problema no entrega información acerca de la zona real en donde podría encontrarse. Por esta razón, es necesario estudiar la influencia de las incertidumbres en las soluciones, con el fin de determinar una aproximación más clara de la zona en donde podría encontrarse el objeto. La figura 4.3 muestra ambos conjuntos solución al considerar las incertidumbres en la medición.

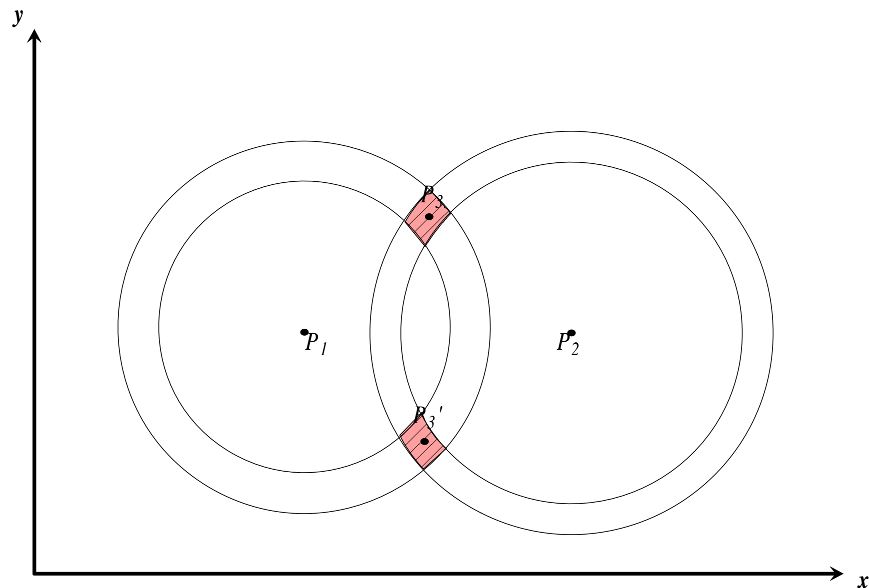


Figura 4.3: Problema en plano cartesiano considerando incertidumbres

La zona achurada representa el espacio solución del problema completo. En ella se observa que existe un rango de tolerancia para ambas soluciones, y que la solución exacta al problema (los puntos P_3 y P_3'), representa sólo un caso particular.

La metodología de resolución de sistemas de ecuaciones con incertidumbre plantea, en primera instancia, la determinación de los puntos solución del problema sin considerar las incertidumbre.

El segundo paso consiste en utilizar técnicas de la programación con restricciones para extender los dominios de las variables de forma consistente. Para ello, es necesario modificar

el modelo original, transformando sus ecuaciones en inecuaciones que consideren la incertidumbre. El nuevo sistema de ecuaciones puede ser escrito de la siguiente manera:

$$
\begin{array}{ll}
x_{1a} = 0 \text{ y } x_{1b} = 0 & \text{(punto } P_1 \text{ fijo)} \\
x_{2a} = 0 \text{ y } x_{2b} = 5 & \text{(punto } P_2 \text{ fijo)} \\
\sqrt{(x_{1a} - x_{3a})^2 + (x_{1b} - x_{3b})^2} \leqslant d_1 + i & \text{(distancia entre } P_1 \text{ y } P_3 \text{ más incertidumbre)} \\
\sqrt{(x_{1a} - x_{3a})^2 + (x_{1b} - x_{3b})^2} \geqslant d_1 - i & \text{(distancia entre } P_1 \text{ y } P_3 \text{ menos incertidumbre)} \\
\sqrt{(x_{2a} - x_{3a})^2 + (x_{2b} - x_{3b})^2} \leqslant d_2 + i & \text{(distancia entre } P_2 \text{ y } P_3 \text{ más incertidumbre)} \\
\sqrt{(x_{2a} - x_{3a})^2 + (x_{2b} - x_{3b})^2} \geqslant d_2 - i & \text{(distancia entre } P_2 \text{ y } P_3 \text{ menos incertidumbre)}
\end{array}
$$

La figura 4.4 muestra el resultado de la extensión de dominios para ambas soluciones.

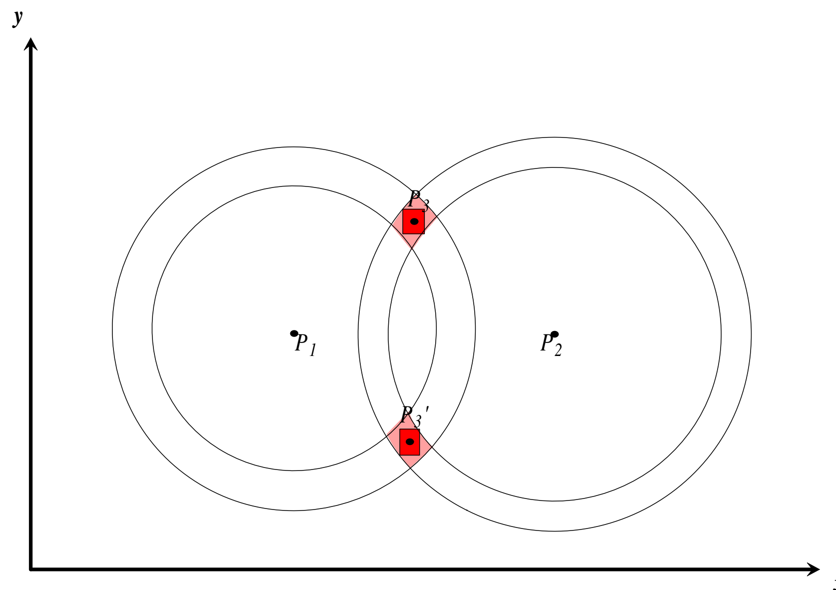


Figura 4.4: Cálculo de caja interior a través de consistencia interna

La caja interna calculada, representa una aproximación inferior al espacio solución, es decir, un subconjunto del espacio de búsqueda original que contiene solamente puntos solución del sistema de ecuaciones. De esta forma es posible afirmar que cualquier punto contenido en la caja interior es solución del problema planteado.

El tercer paso de la metodología consiste en separar cada punto perteneciente a una solución, de sus alternativas existentes en otras soluciones. En otra palabras, se separa el espacio de búsqueda en subespacios, cada uno de los cuales posee sólo una de las soluciones

al problema.

En el problema planteado no existen alternativas para los puntos P_1 y P_2, ya que son considerados fijos. Por el contrario, existen dos alternativas para el tercer punto: P_3 y P_3'. La figura 4.5 muestra la separación de los puntos solución P_3 y P_3' del problema en subespacios disjuntos.

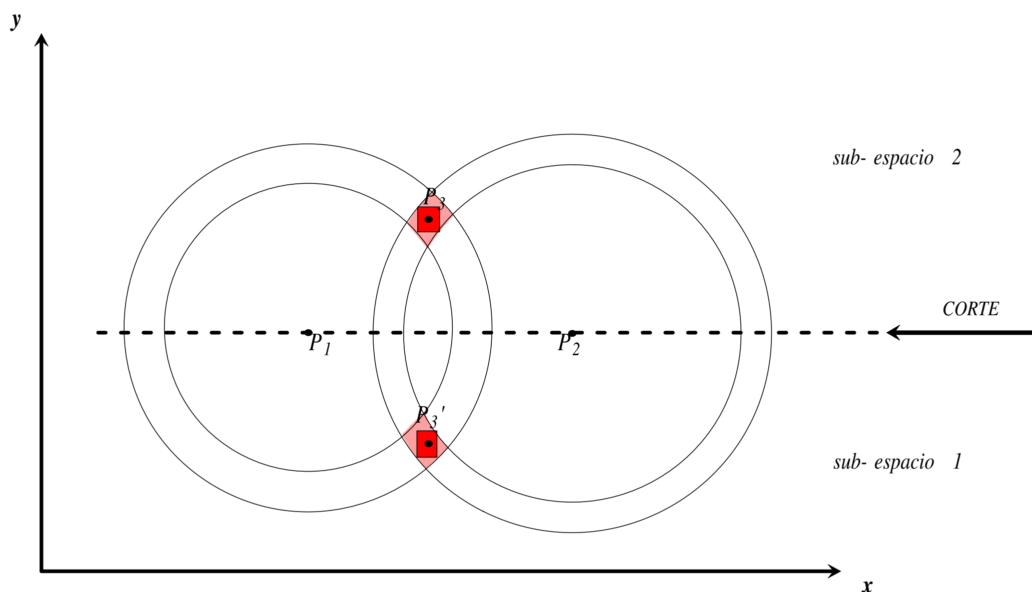


Figura 4.5: Separación de puntos pertenecientes a soluciones diferentes

Una vez independizados los puntos pertenecientes a las soluciones del problema, se procede al cuarto paso de la metodología que consiste en aplicar técnicas de consistencia externa para encontrar una caja exterior que contenga el máximo grado de tolerancia para cada una de las soluciones encontradas.

La figura 4.6 muestra la aplicación de las técnicas de consistencia externa a cada subespacio de búsqueda.

El resultado final de esta metodología es la obtención de una caja interior y una exterior, por cada uno de los puntos pertenecientes a una solución. La caja interior indica una zona de cumplimiento absoluto de las restricciones, es decir, cualquier valor contenido en ella es solución del sistema de ecuaciones planteado (considerando las incertidumbres). Por otro lado, la caja exterior indica una zona de cumplimiento límite, vale decir, no existen puntos fuera de esta zona que cumplan con las restricciones planteadas en el sistema de ecuaciones, aún considerando las incertidumbres.

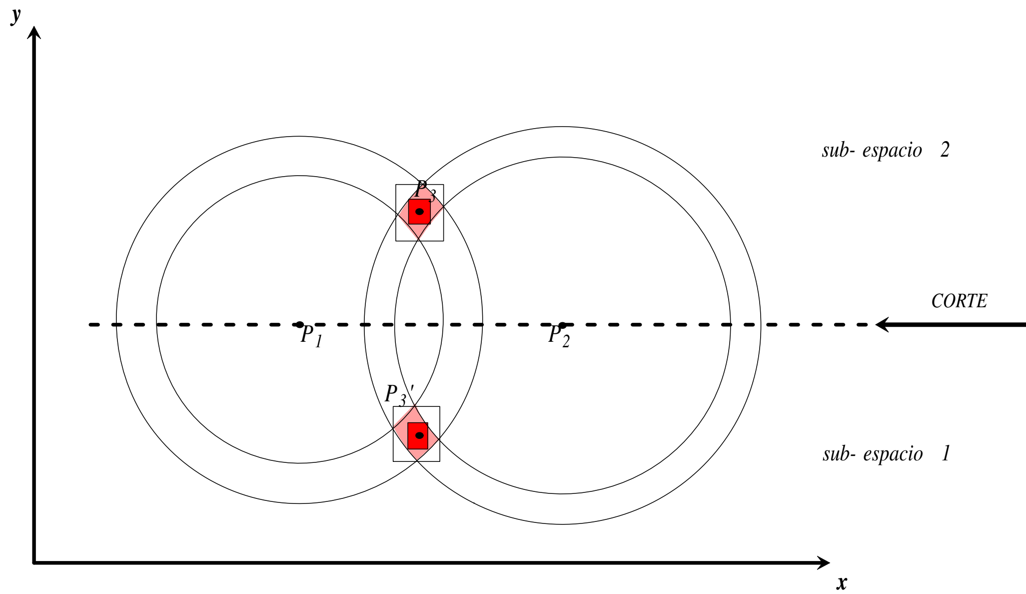


Figura 4.6: Cálculo de cajas exteriores para cada solución

En conclusión, el espacio comprendido entre la caja interior y la caja exterior representa una aproximación de las soluciones del sistema de ecuaciones con incertidumbre.

Finalmente, y como respuesta al problema inicial de encontrar la posición del artefacto emisor de ondas, se puede concluir que existen dos posibles ubicaciones en el terreno. Para cada una de ellas se tiene una caja interior (que marca el lugar más probable donde se puede encontrar el artefacto) y una caja exterior (que marca el límite de la zona en donde se debería buscar.

La ventaja inmediata de esta metodología es la posibilidad de entregar una respuesta más precisa al problema, que refleje las características de incertidumbre involucrada.

4.2. Algoritmo

Para este estudio se consideró un sistema compuesto por p ecuaciones de distancia de la forma $d_(p_i, p_j) \leqslant d_{ij} \pm e_{ij}$, sobre un espacio *d-dimensional* compuesto de n puntos, de los cuales m son fijos.

En primera instancia, se resuelve el sistema de ecuación sin considerar las incertidumbres. Para ello se utilizan bibliotecas especializadas de resolución de sistemas, en particular, ILOG Solver. Para mayor información, en el anexo A se entrega una introducción a las tecnologías

de ILOG Concert e ILOG Solver.

Una vez determinadas las soluciones al problema, se realiza el *cálculo de cajas interiores* (aplicando extensión de dominios i-consistentes) y el *cálculo de cajas exteriores* (aplicando separación de puntos en el espacio). De esta forma se acota la incertidumbre entre ambas cajas.

4.2.1. Cálculo de cajas interiores

Para el cálculo de cajas interiores se aplica el algoritmo de extensión de dominios i-consistentes mostrado en el cuadro 4.1.

```
function icons(V = {x_1, ..., x_d}, D = I_1 × ... × I_d, c)
    for (k = 1; k < d; k++)
        a ← Inf(I_k)
        a ← Sup(I_k)
            I_k ← ]−∞, a]
                2B-consistencia(V, D, !c)
                a_k ← Max(I_k)
            I_k ← [b, ∞[
                2B-consistencia(V, D, !c)
                b_k ← Min(I_k)
            I_k ← [a_k, b_k]
    end for
end function
```

Cuadro 4.1: Algoritmo de extensión de dominios i-consistentes

El algoritmo recibe inicialmente un conjunto de variables que corresponden a las coordenadas de un punto en el espacio *d-dimensional* y sus dominios asociados. Originalmente cada dominio corresponde a un intervalo muy pequeño (o un valor puntual), pues no se consideró las incertidumbres en la resolución del sistema.

A partir de estos intervalos se realiza la extensión de dominios para cada coordenada del punto, una a una, manteniendo las demás coordenadas fijas.

Para extender el dominio en una coordenada particular, considerando una restricción c, se separa el intervalo inicial $[a, b]$ en dos subintervalos disjuntos $]-\infty, a]$ y $[b, \infty[$, manteniendo fijas las demás coordenadas.

La extensión izquierda del intervalo para la coordenada i se obtiene considerando un nuevo CSP con los dominios iniciales para todas las demás coordenadas, y el dominio $]-\infty, a]$

para la coordenada i. A continuación, se aplica una técnica de consistencia externa (por ejemplo, 2B-consistencia), considerando la restricción opuesta a la original (vale decir, $not(c)$ en lugar de c). De esta forma, se obtiene un nuevo intervalo $[c, d]$ con $d \leqslant a$, en donde la restricción opuesta es válida. En conclusión, si la restricción $not(c)$ es válida sólo en el intervalo $[c, d]$, esto implica que la restricción c es válida en el intervalo $[d, a]$, por lo que el nuevo límite inferior para la coordenada i pasa a ser d.

La extensión derecha se realiza en forma análoga, obteniéndose un nuevo intervalo para la coordenada i. De igual forma se extiende el dominio de cada una de las coordenadas, obteniéndose como resultado la caja interior correspondiente.

Dado que existe más de una restricción en el CSP original, es necesario aplicar el algoritmo por cada una de las restricciones del problema, para luego intersectar los intervalos extendidos y obtener la caja interior que cumpla con todas las restricciones del problema. El cuadro 4.2 muestra el algoritmo de extensión general.

```
function iconsgen(V = {x_1, ..., x_d}, D = I_1 × ... × I_d, C = {c_1, ..., c_m})
    for (k = 1; k < m; k++)
        icons(V, D, c_k)
        D_k ← D
    end for
    D ← D_1 ∩ ... ∩ D_m
end function
```

Cuadro 4.2: Algoritmo general de extensión de dominios

Por cada restricción se calcula la extensión de dominios y se obtiene un nuevo dominio D_k. Posteriormente, los D_k son intersectados, obteniéndose una caja interior que por construcción es i-consistente.

4.2.2. Cálculo de cajas exteriores

Debido a que las técnicas de consistencia externa reducen el espacio de búsqueda a un subespacio que contiene todas las soluciones del problema, es necesario independizar los conjuntos solución antes de aplicarlas. La figura 4.7 muestra la caja exterior resultante, sin separación de conjuntos solución, calculada para el problema presentado en la sección 4.1.

Como puede apreciarse en la figura, es necesario independizar los conjuntos solución para obtener mejores aproximaciones en cada una de las posiciones posibles de un punto. Para

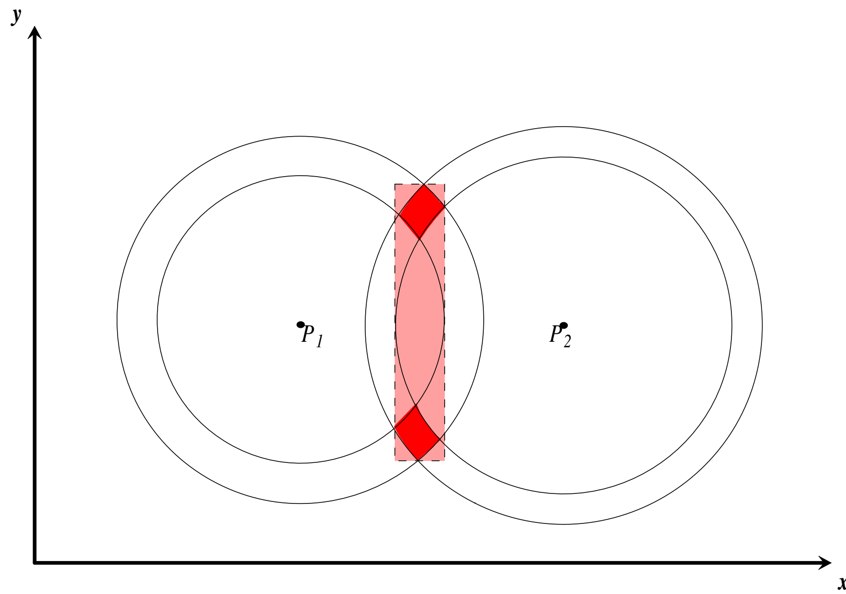


Figura 4.7: Caja exterior calculada sin separación de conjuntos solución

ello, se propone la utilización de una función de verificación de coordenadas, presentada en el cuadro 4.3.

```
function testnaif({A_1, ..., A_r})
    for (k = 1; k < d; k++)
        if (∀ (A_i, A_j) ∈ {A_1, ..., A_r} × {A_1, ..., A_r}, π_k(A_i) ≠ π_k(A_j))
            return k
        end if
    end for
    return 0
end function
```

Cuadro 4.3: Función de verificación de coordenadas

La expresión $\pi_k(A_i)$ representa la proyección del punto A_i en la coordenada k. Esta función recibe el conjunto de puntos a ser separados e investiga, para cada coordenada, si existe más de un punto que comparta el mismo valor. Si todos los puntos tienen un valor distinto para una coordenada en particular, ésta es devuelta como la *coordenada a utilizar para separar el espacio de búsqueda*, en caso contrario, la función retorna una coordenada nula (valor cero).

Asumiendo que la coordenada para la separación no es nula, se aplica la función presentada en el cuadro 4.4, que divide en espacio de búsqueda original en n subespacios disjuntos, cada uno conteniendo uno de los puntos a separar.

function algonaif(E, $\{A_1, \ldots, A_r\}$)
    $p \leftarrow$ testnaif($\{A_1, \ldots, A_r\}$)
    if $(p > 0)$
        sort($\{A_1, \ldots, A_r\}$, p)
        for $(i = 1; i < r; i++)$
            $E_i \leftarrow \prod_{k=1}^{p-1} \pi_k(E) \times [\frac{\pi_p(A_i)+\pi_p(A_{i-1})}{2}, \frac{\pi_p(A_{i+1})+\pi_p(A_i)}{2}] \times \prod_{k=p+1}^{d} \pi_k(E)$
        end for
    end if
end function

Cuadro 4.4: Función de separación de puntos en espacios disjuntos

La función recibe el espacio original y el conjunto de puntos. A partir de la función **testnaif** determina la coordenada de corte. A continuación, todos los puntos son ordenados por el valor de su proyección sobre la coordenada de corte. Finalmente, el espacio es dividido en los puntos medios entre cada par de puntos consecutivos. Por convención se asume que A_0 representa el punto de coordenada $-\infty$ y A_{r+1} el punto de coordenada ∞.

Para aquellos casos particulares, en donde no es posible encontrar una coordenada p en donde realizar la separación de todos los puntos, el algoritmo contempla la separación del espacio en el valor medio del primer par de puntos con coordenadas diferentes. Posteriormente, se analizan los subespacios obtenidos utilizando la función **testnaif**. Este procedimiento se realiza en forma recursiva, hasta que los subespacios resultantes puedan ser divididos a través de **algonaif** o posean sólo un punto solución.

Una vez independizados los espacios que contienen sólo un conjunto solución, se aplica una técnica de e-consistencia para reducir los dominios de las variables y determinar una aproximación externa. El resultado de este procedimiento es una caja exterior que rodea a cada uno de los puntos solución encontrados inicialmente. En otras palabras, por cada uno de los conjuntos solución, se tendrá una aproximación inferior (caja interior) y una superior (caja exterior) que acotan su espacio de solución.

4.3. Resumen del capítulo

En este capítulo se presentó la metodología para el tratamiento de incertidumbres en sistemas de ecuaciones de distancia propuesta por Touati. En primer lugar se presentó una explicación mediante un ejemplo, para familiarizar al lector con los pasos propuestos por la metodología. Posteriormente se mostró el algoritmo en detalle, haciendo hincapié en sus principales estructuras. En lo que sigue se presentará el análisis realizado a esta metodología, sus principales áreas de mejoramiento y los algoritmos propuestos para mejorar e interpretar los resultados obtenidos.

Parte III

Trabajo aplicado

Capítulo 5

Áreas de mejoramiento

Del estudio de la metodología de resolución de sistemas de ecuaciones de distancia con incertidumbre, expuesta en el capítulo 4, se han detectado deficiencias en tres grandes áreas:

1. La determinación cualitativa del tipo configuración de la solución. Es decir, en ella se asume que el sistema de ecuaciones posee solución y además, que los puntos solución se distribuyen de manera *conveniente*.

2. Muy relacionado con la observación anterior, se omite la medición del grado de incertidumbre y su influencia en el cálculo de otros puntos pertenecientes a la solución.

3. Se omite la relación entre puntos pertenecientes a una misma solución, separando todos los valores posibles para un mismo punto, sin considerar que su validez depende, además, de los otros valores asignados a las variables de una solución.

El estudio de estas tres deficiencias permite mejorar el algoritmo de separación de puntos y, de esta forma, obtener mejores aproximaciones en el cálculo de cajas exteriores.

En este capítulo se profundizará en cada una de las deficiencias detectadas, comenzando por la determinación cualitativa del tipo de solución que permite determinar a priori las características generales de las soluciones. Posteriormente se analiza la influencia que tienen las incertidumbres previas en el cálculo posterior de puntos, y la relación que existe entre la validez de un punto y la elección de otro como solución. Finalmente se presentan las dificultades del algoritmo de separación de puntos a través de un ejemplo concreto.

5.1. Determinación cualitativa del tipo de configuración

Los sistemas de ecuaciones de distancia poseen características especiales que les proporcionan ciertas facilidades en el tratamiento de las soluciones. Por ejemplo, es posible estudiar a priori el comportamiento del sistema, basándose en la información de distancias y en la incertidumbre, para determinar, de manera cualitativa, qué tipo de soluciones se deben esperar.

Sea $P_A = (x_1, y_1)$ y $P_B = (x_2, y_2)$ dos puntos de referencia. Sea P_X un tercer punto con relaciones de distancia r_1 y r_2 a los dos primeros, respectivamente. Suponga que existe un cierto grado de incertidumbre en la medición de esta distancia. Es posible clasificar, de forma cualitativa, el tipo de configuración solución en:

1. Un sistema de ecuaciones sin un conjunto solución.

2. Un sistema de ecuaciones con sólo un conjunto solución.

3. Un sistema de ecuaciones con dos o más conjuntos solución disjuntos.

La figura 5.1 muestra estas tres posibilidades para el problema planteado. Para facilitar la compresión de las ideas aquí planteadas, se presenta el ejemplo suponiendo que la distancia desde los puntos de referencia al tercer punto es menor que la distancia entre ambos puntos de referencia. Las observaciones son también válidas cuando los puntos de referencia se encuentran a menor distancia que el tercer punto, obteniéndose una circunferencia incluida en otra.

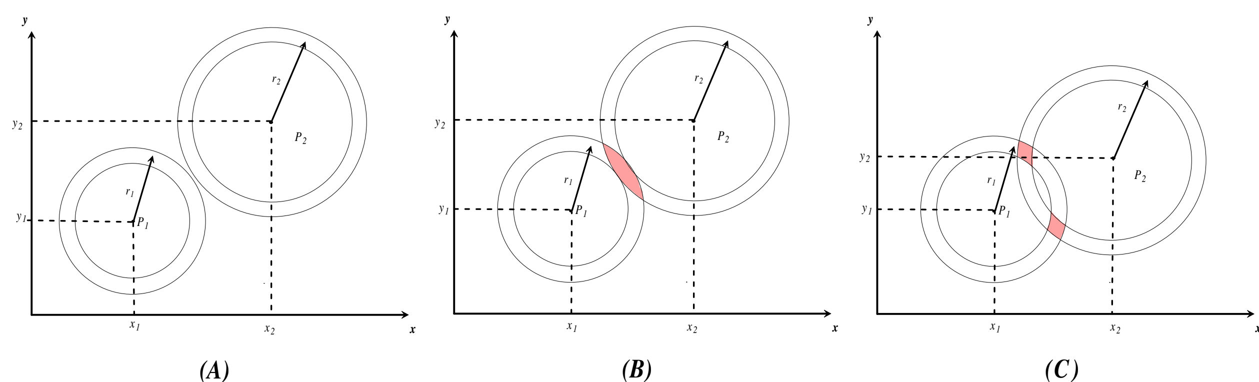


Figura 5.1: Ejemplos de configuración solución

En la letra **A**, se presenta un sistema sin solución, es decir, no es posible encontrar un punto P_x que se encuentre a una distancia r_1 del punto P_A y a distancia r_2 del punto P_B simultáneamente (aún considerando las incertidumbres). La letra **B**, muestra un sistema con un sólo conjunto solución, es decir, cualquiera de los valores pertenecientes a la zona demarcada son solución del sistema. Por último la letra **C**, muestra un sistema con conjuntos solución independientes, es decir, dos zonas del espacio de búsqueda que poseen valores que cumplen todas las restricciones del sistema.

El punto importante a tener en cuenta aquí, es la posibilidad de determinar el tipo de soluciones que se debieran con certeza esperar para el sistema, antes de aplicar un algoritmo de resolución. De esta forma, pueden utilizarse métodos alternativos para cada tipo de sistema, utilizando las características del problema para guiar de mejor forma la metodología.

Existe además dos casos especial de configuración solución, mostrado en la figura 5.2, que pertenece a la clasificación **B** y son llamados *casos degenerados*.

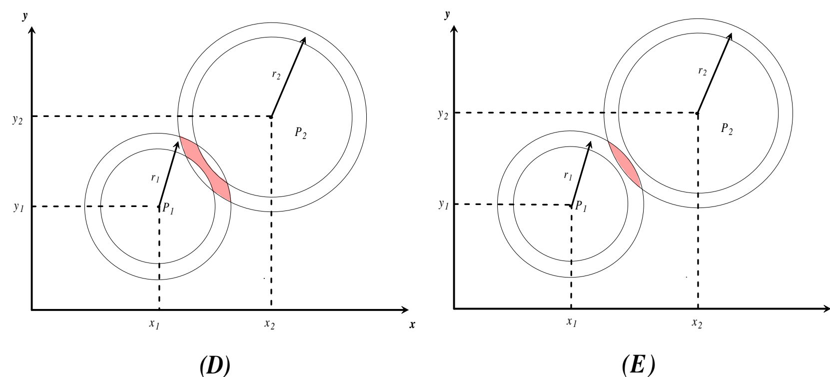


Figura 5.2: Ejemplo de casos degenerados

Estos tipos de configuración solución presentan una mayor dificultar que los otros. En la letra **D**, se muestra un sistema que al calcular las soluciones (sin considerar las incertidumbres) entrega dos puntos independientes, mientras que el sistema con incertidumbre posee sólo un conjunto solución. La letra **E** presenta una dificultad aún mayor, ya que el sistema de ecuaciones sin incertidumbre no posee solución, mientras que el sistema con incertidumbre sí posee un conjunto solución.

5.2. Influencia de la incertidumbre previa

Dado que la determinación de la posición de un punto se realiza en relación a las distancias entre él y los otros puntos de referencia, el grado de incertidumbre en la medición de un punto P influirá sobre todos aquellos puntos determinadas a partir de ecuaciones de distancia al punto P. Esto también influye sobre la configuración solución del problema.

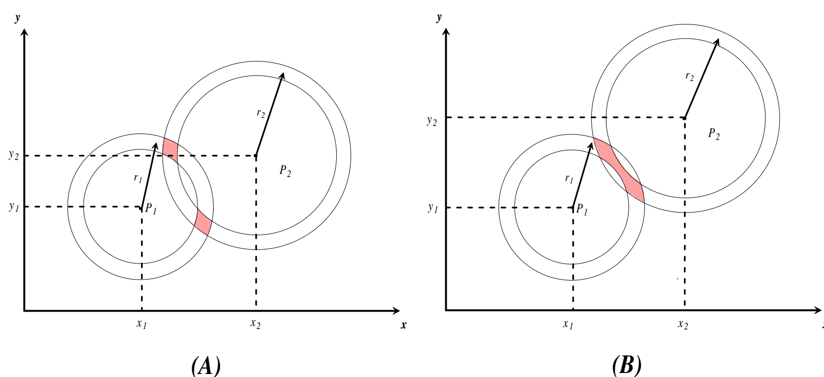


Figura 5.3: Influencia de las incertidumbres previas

La figura 5.3 muestra la configuración solución de un sistema de ecuaciones de distancia con dos puntos de referencia calculados sin incertidumbre (letra **A**). El mismo sistema, considerando que los puntos de referencia han sido calculados a partir de otros puntos de referencia (es decir, poseen un grado de incertidumbre en su medición), es mostrado en la letra **B**. En ella se observa que la configuración solución cambia, producto de la incertidumbre en la posición de los puntos de referencia.

De manera más general, si P_A y P_B son dos puntos de referencia y se desea calcular la posición de un tercer punto P_X, el tipo de configuración solución puede ser aproximado calculando la suma de la incertidumbre en el cálculo del punto de referencia y la incertidumbre en la medición del nuevo punto.

Este enfoque considera que es posible determinar un punto sólo en relación a otros dos puntos de referencia.

5.3. Relación entre los puntos de una solución

Cuando se divide el espacio de búsqueda de dos puntos pertenecientes a soluciones diferentes, todas las soluciones son subdivididas en dos subconjuntos. Esto reduce la cantidad de puntos que debieran ser independizados posteriormente. En estricto rigor, sólo los puntos pertenecientes a las soluciones de cada subconjunto necesitan ser separados de sus pares.

La metodología original considera el cálculo de cajas exteriores para cada conjunto solución. Para ello, se aplica un algoritmo de separación de puntos para independizar las soluciones y evitar la influencia de un conjunto solución en el cálculo de la caja exterior de otro conjunto solución.

En la sección 4.2 se vio que para obtener mejores aproximaciones de cajas exteriores era necesario separar, para cada punto a determinar, el espacio de búsqueda en dos o más subespacios independientes que contengan sólo un conjunto solución. Si bien este enfoque parece correcto, existen sin embargo casos particulares de solución, donde la influencia de sus puntos es parcial.

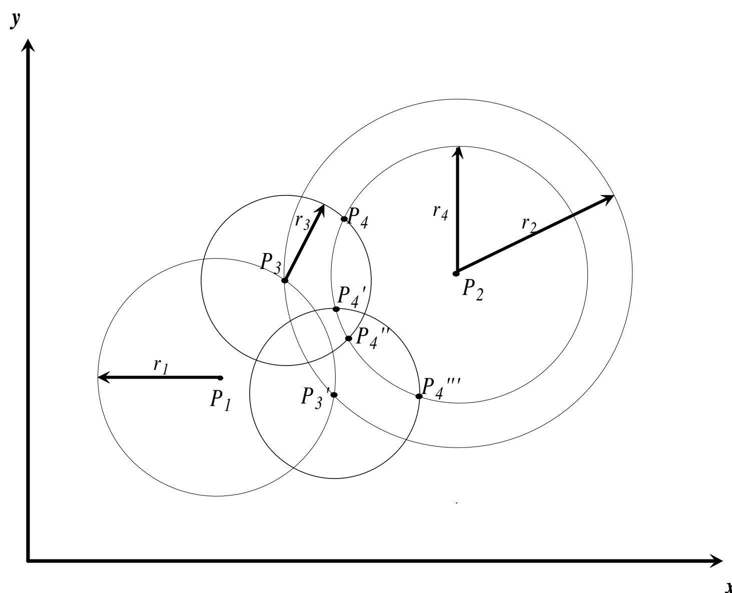


Figura 5.4: Influencia parcial en la determinación de puntos solución

La figura 5.4 muestra el problema de ubicar los puntos P_3 y P_4 en un plano cartesiano, dados los puntos de referencia P_1 y P_2 fijos. Sea $r_1 = d(P_1, P_3)$ la distancia entre el primer y el tercer punto, $r_2 = d(P_2, P_3)$ la distancia entre el segundo y el tercer punto, $r_3 = d(P_2, P_4)$

la distancia entre el segundo y el cuarto punto y $r_4 = d(P_3, P_4)$ la distancia entre el tercer y el cuarto punto. Se asume que todas las mediciones de distancia posee un cierto grado de incertidumbre.

La solución del problema para el tercer punto muestra dos posibles ubicaciones: P_3 y P'_3. Además, el cuarto punto posee cuatro ubicaciones posibles: P_4, P'_4, P''_4 y P'''_4.

La figura muestra que existe interferencia entre los punto P'_4 y P''_4, debido a que comparten un área del espacio en común. Sin embargo, esta influencia podría ser calificada como *parcial*, debido a que P'_4 es un punto solución del problema sólo si se considera que P_3 es la ubicación del tercer punto, mientras que P''_4 es punto solución sólo si P'_3 es la ubicación del tercer punto.

Esta observación tiene una fuerte implicancia en el algoritmo de separación utilizado por la metodología, ya que significa que no es necesario separar los cuatro conjuntos solución (ubicaciones posibles) del cuarto punto, debido a que la separación del espacio de búsqueda del tercer punto provoca automáticamente una separación de las soluciones en dos subconjuntos conteniendo dos posiciones posibles para el cuarto punto.

5.4. Dificultades del algoritmo de separación de puntos

Este algoritmo se basa en la función presentada en el cuadro 4.3. El objetivo de esta función es encontrar una coordenada donde no exista más de un punto con el mismo valor.

Primeramente se seleccionan todos todas las posiciones diferentes para un mismo punto perteneciente a las soluciones. Luego, el conjunto de puntos es enviado a la función *testnaif()*. Esta función examina la primera coordenada (proyección del punto sobre un eje) donde no existen valores coincidentes dos o más puntos. La coordenada es retornada como resultado de la llamada.

Si existe una coordenada que cumpla con la condición, se ordena el conjunto de puntos por el valor de su proyección sobre esta coordenada y se divide el espacio en subespacios independientes, cada uno conteniendo sólo una de las posiciones posibles para el punto dado, como se muestra en el cuadro 4.4.

Se han encontrado fuertes limitantes para este enfoque. En primer lugar, la separación del espacio de búsqueda en subespacios independientes se realiza *por punto* y no *por solución*. En la sección 5.3 se muestra el ejemplo de un sistema formado por cuatro puntos. La figura 5.4 muestra las posiciones factibles para el tercer y cuarto punto. En el ejemplo, la separación del

espacio de búsqueda del tercer punto divide el conjunto de soluciones en dos subconjuntos. En otras palabras: *cuando el espacio de dos puntos es dividido, el conjunto de soluciones es también dividido, por lo tanto, sólo aquellas posiciones pertenecientes al mismo subconjunto de soluciones requerirán separación para cada punto en particular.*

En segundo lugar, la determinación de una coordenada con diferentes valores, para todas las posiciones factibles, de un determinado punto no garantizan la correcta obtención de cajas exteriores. La figura 5.5 muestra la aplicación del algoritmo *algonaif();* para un problema de tres puntos (dos fijos) y dos soluciones factibles.

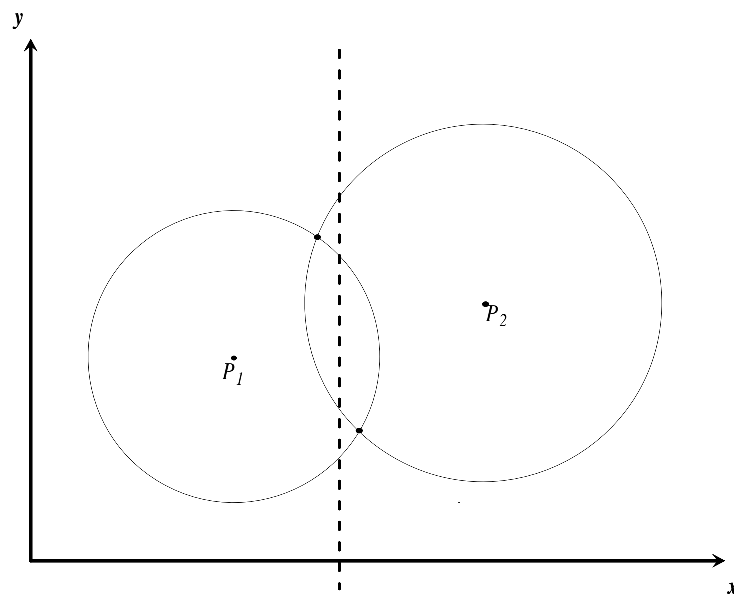


Figura 5.5: Aplicación del algoritmo de separación de puntos

Ambas soluciones no poseen igual valor en su coordenada X, por lo que la respuesta de la función *testnaif();* entrega ésta como coordenada de corte. La separación de ambos puntos se realiza en el punto medio entre sus proyecciones sobre el eje X.

La figura 5.6 muestra el resultado del cálculo de cajas exteriores a partir de la división realizada por *algonaif();*. En ella se observa que ambas soluciones poseen problemas en el cálculo de cajas exteriores. En este ejemplo, un corte a través del eje Y habría entregado una correcta aproximación para ambos conjuntos solución.

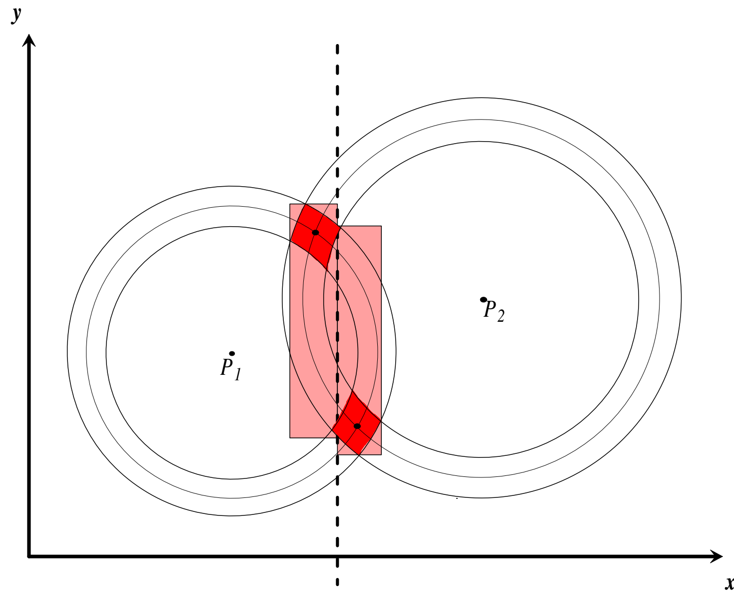


Figura 5.6: Cálculo de cajas exteriores a partir del espacio dividido

5.5. Conclusión del capítulo

En este capítulo se presentaron las posibles áreas de mejoramiento, haciendo énfasis en la influencia de las incertidumbres en la aproximación de soluciones, y las dificultades del actual algoritmo de separación de puntos en el cálculo de cajas exteriores. En el próximo capítulo se presentan dos nuevos algoritmos que mejoran los resultados de la metodología, lo que se debe a la importancia de la separación en el cálculo de cajas exteriores y que motivará la creación de un nuevo algoritmo de separación de puntos y de análisis posterior de soluciones, con el fin de determinar errores en la etapa de división del espacio.

Capítulo 6

Mejoras Propuestas

Tanto para la resolución de los sistemas de ecuaciones e inecuaciones, como para la aplicación de las técnicas de consistencia interna y externa se utilizaron las bibliotecas del paquete $ILOG - Solver$ e $ILOG - Concert$. Una introducción al uso de estas herramientas puede ser consultada en el apéndice A. Información adicional puede ser obtenida en [Nev02b] y [Nev02a], o en los manuales de usuario [ILO00b] y de referencia [ILO00a] del paquete. ç

Las mejoras propuestas se estructuran de la siguiente manera: en primer lugar se estudiará las características de los sistemas de ecuaciones de distancia, que pueden ser utilizadas en la implementación de un algoritmo de clasificación de problemas. Posteriormente se presentará un algoritmo de separación de puntos que permite dividir un espacio inicial en dos o más subespacios disjuntos que contienen un solo punto. Este algoritmo será utilizado por el algoritmo de separación de soluciones, que utiliza una estructura de árbol para independizar efectivamente las soluciones. Finalmente se expone el algoritmo de análisis de soluciones, seguida por la metodología propuesta y un conjunto de resultados preliminares.

6.1. Detección y clasificación de tipos de problemas

A continuación se propone una metodología de clasificación cualitativa del tipo de configuración solución. El objetivo es clasificar los problemas según el tipo de solución esperada, para decidir, a priori, la forma en la cual deben ser interpretadas las soluciones y la forma en la cual deberán separarse los puntos.

Dado un problema de satisfacción de restricciones en dominios continuos $P = (X, D, C)$, definido por un conjunto de puntos $P_A = (x_1, y_1)$ y $P_B = (x_2, y_2)$ fijos y P_X variable, con

restricciones de distancia $d(P_X, P_A) = r_1 \pm i_1$ y $d(P_X, P_B) = r_2 \pm i_2$ a los dos primeros, respectivamente; y dado d_{AB} la distancia entre los puntos P_A y P_B, si $d_{AB} > (r_1 + i_1)$ y $d_{AB} > (r_2 + i_2)$ se tiene que:

1. Si $(r_1 + i_1) + (r_2 + i_2) < d_{AB}$, entonces el sistema no tiene solución (caso **A**, figura 5.1).
2. Si $((r_1 - i_1) + (r_2 - i_2) < d_{AB}) \wedge ((r_1 + i_1) + (r_2 + i_2 > d_{AB})$ y $((r_1 + i_1) + (r_2 - i_2) < d_{AB}) \wedge ((r_1 - i_1) + (r_2 + i_2) < d_{AB})$, entonces existe sólo un conjunto solución al problema. (caso **B**, figura 5.1).
3. Si $(r_1 - i_1) + (r_2 - i_2) > d_{AB}$ entonces existe más de un conjunto solución independiente (caso **C**, figura 5.1).
4. Finalmente, si $((r_1 - i_1) + (r_2 - i_2) < d_{AB}) \wedge ((r_1 + i_1) + (r_2 + i_2 > d_{AB})$ y $((r_1 + i_1) + (r_2 - i_2) > d_{AB}) \vee ((r_1 - i_1) + (r_2 + i_2) > d_{AB})$, entonces existe un caso degenerado (figura 5.2).

Estas observaciones pueden ser fácilmente adaptadas para el caso en que $d_{AB} < (r_1 + i_1)$ o $d_{AB} < (r_2 + i_2)$, de tal forma que es posible estimar las posibles soluciones a partir de la información del problema.

No es el propósito de esta tesis el generalizar el conjunto de observaciones a más de dos dimensiones, sino sólo mostrar que es posible, bajo ciertas condiciones, conocer de forma cualitativa las características de la configuración solución.

Estas proposiciones ayudan a determinar cómo deben ser interpretadas las soluciones del problema, por ejemplo, en el caso de conjuntos solución independientes (caso **C**), éstos deberán ser separados antes de aplicar una técnica de consistencia externa. Por otro lado, si existe sólo un conjunto solución (o un caso degenerado), las técnicas de consistencia externa deberán envolver un sólo conjunto de solución, aún cuando el problema posea más de un punto solución y se haya calcular más de una caja interior.

6.2. Un algoritmo de separación de puntos para restricciones de distancia

Para aplicar una técnica de consistencia externa que filtre el dominio de las variables, es necesario separar en espacios independientes las posiciones posibles (conjuntos solución) de

un mismo punto. Si esta separación se realiza de un modo incorrecto, las cajas externas calculadas a partir de los subespacios obtenidos serán también incorrectas. El algoritmo general de separación de puntos mostrado en el cuadro 6.1 reduce los casos donde la separación de puntos provoca errores en el cálculo de cajas exteriores, de la metodología mostrada en el capítulo 4.

Sin perjuicio de lo anterior y una vez calculada la caja exterior para un conjunto solución, es posible examinar la relación existente entre *el punto de corte del espacio de búsqueda* (resultante del algoritmo de separación) y los *límites de la caja exterior*. Si ambos valores coinciden, existe una alta probabilidad de que la separación del espacio se haya realizado de forma incorrecta.

```
function sepa(H = {A_1, ..., A_m}, E)
    if (|H| > 1)
        point ← newplansepa(H, &coord);
        for (i = 1; i < m; i++)
            if (A_i[coord] <point)
                H_inf = H_inf + {A_i}
            else
                H_sup = H_sup + {A_i}
            end if
        end for
        E_inf = E − (point, +∞)
        E_sup = E − (−∞, point)
        sepa(H_inf, E_inf)
        sepa(H_sup, E_sup)
    else
        A_1 = E
    end if
end function
```

Cuadro 6.1: Algoritmo general de separación de puntos

El algoritmo utiliza la función *newplansepa();* que recibe el conjunto de puntos a separar (**H**) y entrega la coordenada (**coord**) donde se encuentra la mayor distancia entre dos proyecciones de puntos consecutivas y el punto medio (**point**) entre ambas proyecciones (lugar en donde se deberá aplicar el corte). Para realizar esto, los puntos son ordenados por sus proyecciones sobre cada una de las coordenadas. Posteriormente, se calculan las distancias entre dos proyecciones consecutivas y se almacena la mayor distancia encontrada. A continuación la función retorna la coordenada con la mayor distancia y el punto en donde se

realizará el corte.

La función *sepa();* utiliza esta información para dividir el espacio de búsqueda original en dos subespacios independientes: E_{inf} y E_{sup}. Para cada punto $A_i \in H$ se calcula el valor de la coordenada **coord**. Si el valor de A_i es mayor que el valor de **coord**, entonces A_i es almacenado en el subconjunto superior H_{sup}, en caso contrario es almacenado en H_{inf}.

Una vez separados los puntos en dos subconjuntos independientes, se divide el espacio de búsqueda original y se generan los subespacios E_{inf} y E_{sup}. El primero se genera restando al espacio de búsqueda original, todos aquellos puntos ubicados desde el punto de corte en adelante. El segundo subespacio se genera restando al espacio original todos aquellos puntos menores que el punto de corte.

Finalmente, para cada subconjunto de puntos se realiza una llamada a la función de separación original, suministrando el nuevo conjunto de puntos a separar y el nuevo espacio de búsqueda generado. Estas llamadas recursivas se realizan hasta que en cada subespacio de búsqueda sólo exista un punto solución.

6.3. Algoritmo de separación de soluciones

A partir de las observaciones realizadas en la sección 5.2, se implementó una estructura de árbol para separar soluciones. Esta estructura permite diferenciar entre aquellos puntos que interfieren en el cálculo de cajas externas y aquellos que no. La idea principal es aplicar una función de pre-separación (antes de la aplicación del algoritmo de separación), para seleccionar sólo aquellos puntos solución que aún son válidos al considerar las separaciones de puntos realizadas anteriormente. El cuadro 6.2 muestra la función de separación.

```
function presepa(S = {S_1, ..., S_n}, E = {E_1, ...}, pos)
    for (i = 1; i < n; i++)
        if (S_i[pos] ⊂ E_pos)
            S_aux = S_aux + {S_i}
    S = S_aux
    return 0
end function
```

Cuadro 6.2: Aplicación de una función de separación

Esta función recibe tres parámetros:

1. El conjunto de todas las soluciones al problema.

2. Los subespacios generados hasta el momento de la llamada, para todos aquellos puntos previamente separados.

3. La identificación del puntos que deberán ser separados.

Para cada solución, la función analiza si las posiciones de sus puntos se encuentran dentro de los subespacios generados anteriormente. Todas aquellas soluciones que cumplan con esta condición deberán ser consideradas en la separación. La función modifica el conjunto inicial de soluciones, manteniendo sólo aquellas que interferirán en el cálculo posterior de cajas exteriores.

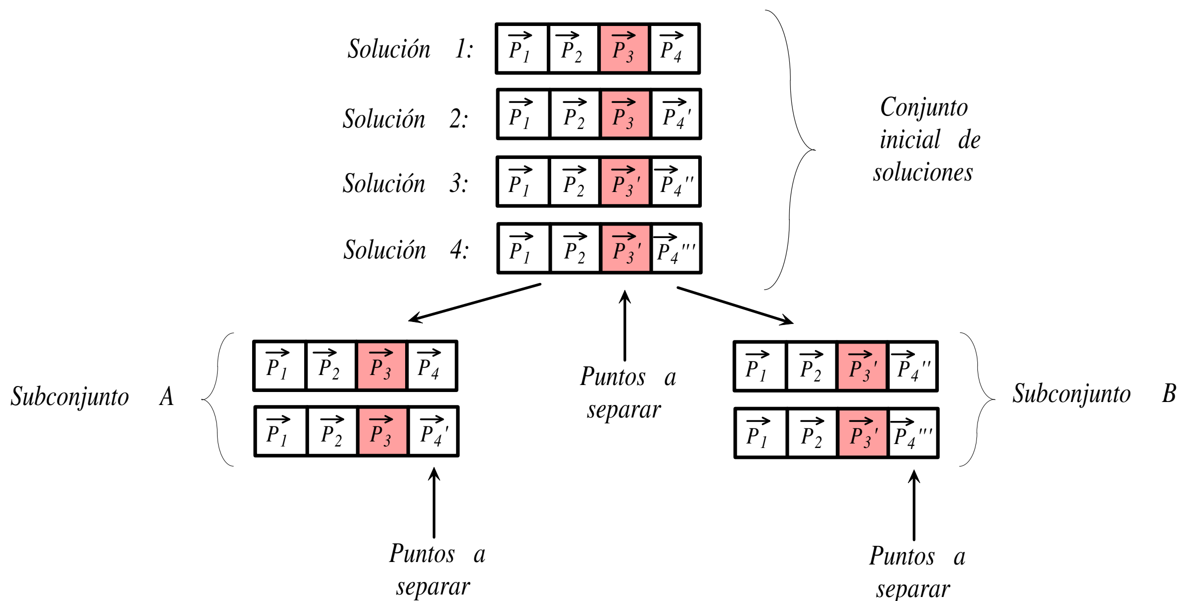


Figura 6.1: Ejemplo de aplicación de una estructura de árbol

La figura 6.1, muestra un ejemplo de la aplicación de una estructura de árbol para el problema mostrado en la figura 5.4. Existen cuatro soluciones factibles, mostradas en la parte superior del árbol. Al separar el espacio de búsqueda del tercer punto en dos subespacios independientes, las soluciones quedan separadas en dos subconjuntos, cada uno conteniendo las posiciones factibles para el cuarto punto (próximo a separar). Desde este punto de vista, no es necesario separar todas las posiciones posibles para el cuarto punto, ya que sólo aquellas pertenecientes al mismo subconjuntos solución producirán problemas en el cálculo

de cajas exteriores. A continuación, a cada subconjunto de soluciones se aplica la separación de puntos, mostrada en la sección 6.2 para aquellos que aún poseen más de una posición factible.

Finalmente cada solución posee un espacio de búsqueda asociado, en donde sólo ella es factible. De esta forma, la aplicación de un algoritmo de filtraje por consistencia externa entregará mejores aproximaciones para las cajas exteriores.

En el ejemplo se muestra el hecho que, si bien los puntos P_4' y P_4'' comparten un área común en el plano (por lo que la separación de su espacio no sería posible), la validez de sus posiciones queda determinada por la elección del tercer punto. Luego, si el espacio perteneciente a este último es separado, las posibles posiciones para el cuarto punto contemplarán a P_4' ó P_4'', pero no ambos al mismo tiempo.

6.4. Algoritmo de análisis de soluciones

Si bien los resultados de los algoritmos propuestos reducen los errores en el cálculo de cajas exteriores, aún es posible obtener resultados incorrectos. Para detectar esta situación, se implementó un algoritmo de análisis posterior de soluciones, con el fin de detectar posibles errores en la etapa de separación. Este algoritmo compara los límites de la caja exterior calculada para cada conjunto solución, con los límites del subespacio al cual pertenece dicha solución. Si ambos valores coinciden, se presume que el conjunto solución poseía soluciones que excedían los límites del subespacio, lo que significa que el algoritmo de separación de puntos realizó un corte incorrecto.

Por otra parte, existe la posibilidad que un punto en particular posea problemas en el límite superior, para ciertas soluciones, y en el límite inferior, para otras. En estos casos, es muy probable que los puntos separados no sean realmente separables, ya que las cajas exteriores calculadas para ambos subespacios exceden precisamente el mismo límite.

A través de este algoritmo se evaluó el desempeño de ambas metodologías, asumiendo que una caja exterior coincidente con el límite del subespacio evidencia un problema de separación.

6.5. Metodología

La metodología propuesta se puede resumir en el siguiente algoritmo:

- Aplicar un algoritmo de análisis previo, que permita clasificar el problema en aquellos que cumplen con los supuestos de separación de puntos solución y aquellos que producirán errores en el cálculo de cajas exteriores.
- Determinar las soluciones al problema sin considerar la influencia de las incertidumbres.
- Incluir las incertidumbres en el modelo, reemplazando las ecuaciones originales por sus inecuaciones correspondientes y, por cada solución encontrada, extender los dominios de las variables de forma i-consistente para determinar una caja interior de tolerancia.
- Utilizar el algoritmo de separación de soluciones, mostrado en la sección 6.3, para dividir el espacio de búsqueda inicial, de manera que cada subespacio generado posea sólo una solución válida.
- Para cada subespacio generado y por cada punto perteneciente a la solución contenida en él, aplicar técnicas de consistencia externa con el fin de reducir el espacio hasta encontrar la mínima caja exterior que contiene a cada conjunto solución.
- Aplicar un algoritmo de análisis de cajas exteriores, que compare los límites de los subespacios generados y los límites de las cajas exteriores calculadas.

El objetivo es que, una vez clasificado el problema, se detecte la forma más eficiente de resolver, se determine el algoritmo de separación a utilizar y la interpretación que se le deberá dar a los resultados.

Por otra parte, con la estructura de árbol se obtiene una separación de *soluciones*. Esto permite reducir los errores en el cálculo de cajas exteriores, ya que evita la separación de puntos, pertenecientes a distintas soluciones, que no producen interferencias mutuas. Además, con el algoritmo de separación de puntos se reducen los problemas de separaciones incorrectas cuando los puntos se encuentran muy próximos en una coordenada particular pero son aún separables en otra, detectados en el algoritmo inicial.

6.6. Resultados preliminares

Se seleccionaron cuatro casos particulares para mostrar la aplicación de la metodología mejorada. El primero corresponde al problema de cuatro puntos en el plano mostrado en la figura 5.4. El segundo corresponde a un problema simplificado de conformación molecular

de una proteína. Por último, los otros dos problemas corresponde a la aplicación de la metodología para un problema más general.

6.6.1. Sistema de ecuaciones de distancia en el plano

El modelo del problema está formado por el sistema de cuatro ecuaciones mostrado a continuación:

$$\begin{aligned}
\sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2} &= 4{,}0 \quad (e_1 = 0.1)\\
\sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2} &= 9{,}1 \quad (e_2 = 0.2)\\
\sqrt{(x_4 - x_3)^2 + (y_4 - y_3)^2} &= 7{,}1 \quad (e_3 = 0.2)\\
\sqrt{(x_4 - x_2)^2 + (y_4 - y_2)^2} &= 4{,}0 \quad (e_4 = 0.1)
\end{aligned}$$

$$D_{x_1} = 1,\ D_{y_1} = 0,\ D_{x_2} = 10,\ D_{y_2} = 0,\ D_{x_i} = D_{y_i} = [-100, 100]\ \ \forall i = \{3, 4\}$$

En el cuadro 6.3 se muestran los resultados obtenidos al aplicar el algoritmo actual para resolver el sistema de ecuaciones de distancia. En él, se destacan cuatro valores pertenecientes al límite de las cajas exteriores.

	x_3	y_3	x_4	y_4
Sol. 1	2.58813	3.72564	9.7868	3.86402
i-box	[2.4881, 2.6881]	[3.7256, 3.7363]	[9.7858, 9.787]	[3.8640, 3.8650]
e-box	[2.4871, 2.7]	[3.6395, 3.8094]	[**9.7868**, 9.9986]	[3.8115, 3.9751]
Sol. 2	2.58812	3.72564	7.32326	-1.69825
i-box	[2.4788, 2.6976]	[3.7246, 3.7266]	[7.2804, 7.3636]	[-1.6982, -1.6663]
e-box	[2.4755, 2.7]	[3.6395, 3.8094]	[7.1960, **7.3232**]	[-1.8142, -1.4590]
Sol. 3	2.58812	-3.72564	9.7868	-3.86402
i-box	[2.4881, 2.6881]	[-3.7266, -3.7246]	[9.7868, 9.7878]	[-3.8640, -3.8630]
e-box	[2.4755, 2.6898]	[-3.8094, -3.6395]	[9.5743, **9.7868**]	[-3.9163, -3.7377]
Sol. 4	2.58813	-3.72564	7.32326	1.69825
i-box	[2.4788, 2.6976]	[-3.7256, -3.7165]	[7.2899, 7.3647]	[1.6972, 1.6992]
e-box	[2.4755, 2.7]	[-3.8094, -3.6395]	[**7.3232**, 7.4591]	[1.5752, 1.9276]

Cuadro 6.3: Resultados obtenidos con el algoritmo actual

Es evidente que estos valores no pueden ser correctos, ya que incluso son más restrictivos que los encontrados para las cajas interiores. Dada las características del problema, una caja exterior nunca podría tener un espacio asociado menor que una caja interior calculada para el mismo conjunto solución.

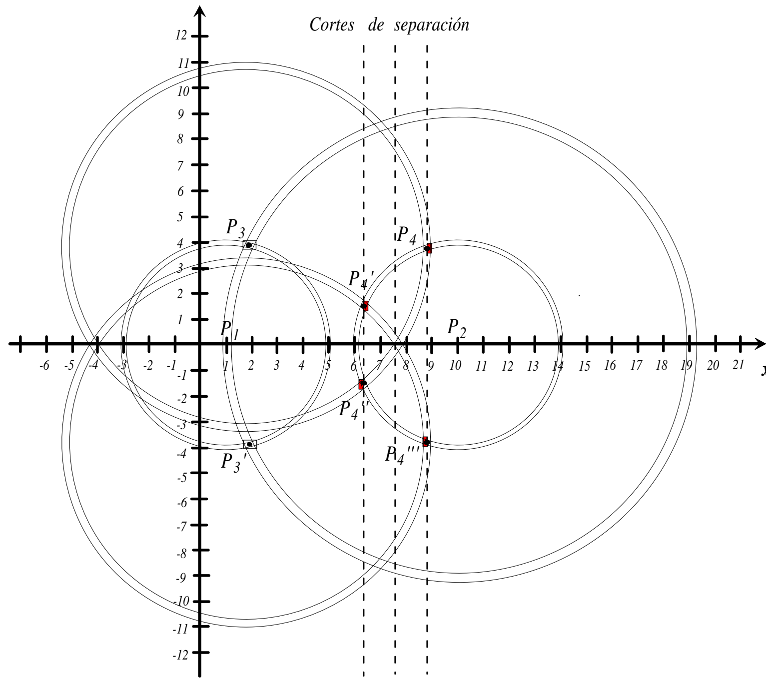


Figura 6.2: Resultados del algoritmo actual, en el plano cartesiano

La figura 6.2 muestra gráficamente los resultados obtenidos, detallando los cortes de separación del espacio para el cuarto punto. En ella se observa que pequeñas variaciones en los valores de las coordenadas de un punto pueden ser determinante en la separación (es el caso del cuarto punto, para el cual, el algoritmo a seleccionado la coordenada X, para separar verticalmente el espacio). Las cajas exteriores calculadas para cada solución poseen un límite coincidente con el límite del subespacio.

	x_3	y_3	x_4	y_4
Sol. 1	2.58812	3.72564	9.7868	3.86402
e-box	[2.4755, 2.7]	[3.6395, 3.8094]	[9.5743, 9.9986]	[3.7377, 3.9751]
Sol. 2	2.58813	3.72564	7.32326	-1.69825
e-box	[2.4755, 2.7]	[3.6395, 3.8094]	[7.1960, 7.4591]	[-1.9276, -1.459]
Sol. 3	2.58813	-3.72564	9.7868	-3.86402
e-box	[2.4755, 2.7]	[-3.8094, -3.6395]	[9.5743, 9.9986]	[-3.9751, -3.7377]
Sol. 4	2.58813	-3.72564	7.32326	1.69825
e-box	[2.4755, 2.7]	[-3.8094, -3.6395]	[7.1960, 7.4591]	[1.459, 1.9276]

Cuadro 6.4: Resultados obtenidos con el algoritmo propuesto

El cuadro 6.4 muestra los resultados obtenidos por el nuevo algoritmo propuesto, aplicado

al mismo problema. Los valores correspondientes a las cajas interiores se han omitido, ya que son los mismos valores mostrados anteriormente.

La principal diferencia entre ambos radica en el cálculo de cajas exteriores. Mientras el primer algoritmo realiza un corte a través del eje X (dado que los puntos no poseen igual valor en sus coordenadas), el segundo busca la mayor distancia entre proyecciones consecutivas, realizando cortes en el eje Y (como se muestra en la figura 6.3). El resultado final es una correcta aproximación de la caja exterior que contiene al conjunto solución.

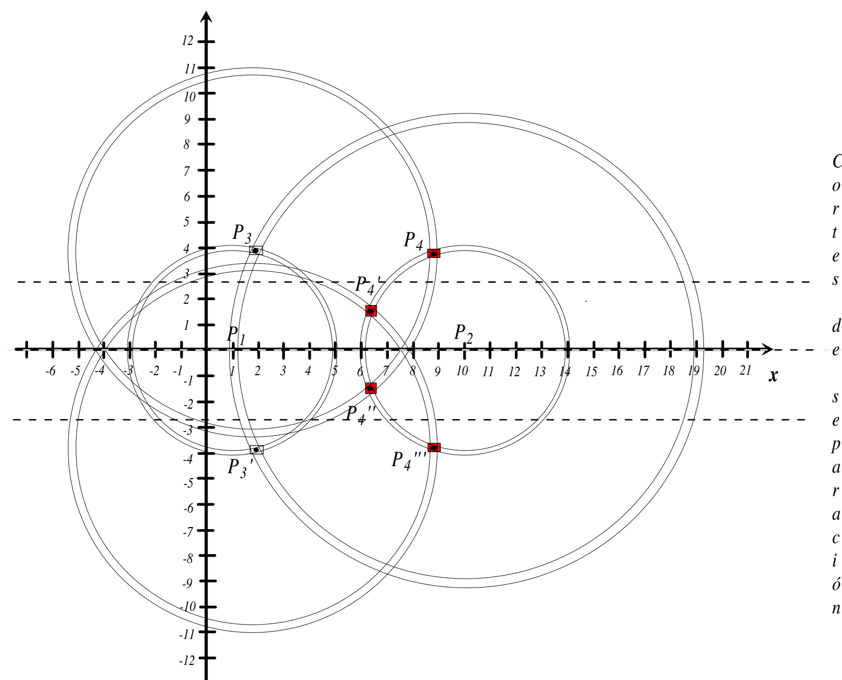


Figura 6.3: Resultados del algoritmo propuesto, en el plano cartesiano

Una observación adicional es la siguiente: La determinación incorrecta de la caja exterior para un conjunto solución dado provoca resultados incorrectos en la determinación de otras cajas exteriores, aún cuando la separación de puntos para estas últimas se haya realizado en forma correcta.

6.6.2. Determinación de la estructura de una molécula

Las proteínas juegan un rol fundamental en el control del metabolismo, y la forma de la proteína es un factor esencial para su funcionamiento. En los libros de bioquímica se utiliza comúnmente la analogía de *la llave y el candado* para representar la importancia que tiene

su estructura. A esto se debe que en mucha áreas de investigación, como la biotecnología o la medicina, la determinación de la estructura molecular de una proteína juegue un rol fundamental. Los avances en genética han permitido conocer cerca de 100.000 secuencias de proteínas, pero sólo un 10 % de ellas posee estructura conocida. Una de las principales dificultades en la determinación de su estructura es su alto nivel combinatorio y la incertidumbre en la medición de las distancias. Esta incertidumbre se debe a que las mediciones se realizan en forma experimental.

Básicamente, una proteína es una macromolécula formada por un elevado número de aminoácidos distribuidos en el espacio de una manera particular. Para determinar su estructura, existen diferentes métodos que intentan simplificar el problema para obtener una aproximación a la solución. Uno de esto métodos consiste en determinar experimentalmente, a través de resonancia magnética nuclear, la distancias entre sus aminoácidos constituyentes, y resolver un sistema de ecuaciones de distancia.

El siguiente es un ejemplo simplificado del problema de determinar la estructura de una molécula, conociendo una aproximación de las distancia entre sus átomos constituyentes.

Se tienen seis átomos A_1, A_2, A_3, A_4, A_5 y A_6, y la información de distancias mostrada en el cuadro 6.5.

	A_1	A_2	A_3	A_4	A_5	A_6
A_1	–	$\sqrt{2}$	2	–	2	2
A_2	$\sqrt{2}$	–	$\sqrt{2}$	2	$\sqrt{2}$	–
A_3	2	$\sqrt{2}$	–	$\sqrt{2}$	2	2
A_4	–	2	$\sqrt{2}$	–	$\sqrt{2}$	$\sqrt{2}$
A_5	2	$\sqrt{2}$	2	$\sqrt{2}$	–	–
A_6	2	–	2	$\sqrt{2}$	–	–

Cuadro 6.5: Información de distancia entre átomos

El objetivo es determinar, a partir de la información de distancias dada, la estructura espacial de la molécula, considerando una incertidumbre en la medición que no supera el 5 %. Para ello se considera un modelo de seis puntos en $\mathbb{R}^3$, y un conjunto de ecuaciones de distancia con incertidumbre similar al del problema anterior.

Debido a que el problema posee un conjunto elevado de soluciones, se presenta en el cuadro 6.6 un resumen de los resultados de ambos algoritmos.

Total de soluciones encontradas	**108**
Total de soluciones diferentes	**32**
Errores en cálculo de ebox (algoritmo original)	**114**
Errores en cálculo de ebox (algoritmo propuesto)	**32**
Porcentaje de reducción de errores	**72 %**
Puntos problemáticos	**{3, 5}**

Cuadro 6.6: Resumen de resultados obtenidos por ambos algoritmos

En ambos casos se encontraron igual número de soluciones y el cálculo de cajas interiores no sufrió modificaciones. Por otra parte, de los 1152 valores calculados para cajas exteriores (correspondientes a 32 soluciones que involucran 6 puntos cada una, con 3 coordenadas por punto y 2 límites por coordenada), se obtuvo una disminución significativa de problemas de límites. Cada error reportado[1] corresponde a una coincidencia entre el punto de corte del espacio (de la etapa del algoritmo de separación de puntos), y la caja exterior calculada para ese subespacio.

Es importante señalar, que los errores encontrados en el algoritmo propuesto corresponden al cálculo de las cajas exteriores para el tercer y quinto átomo, y que ambos límites (superior como inferior), poseían los mismo errores. Por ello, es razonable suponer que la configuración solución no permitía la separación de ambos conjuntos solución en subespacios independientes. El cuadro 6.7 muestra el detalle de errores encontrados para ambos puntos.

Átomo	**3**	**5**
Ubicación del error	**2.2877**	**2.03385**
Errores en límite inferior	**8**	**8**
Errores en límite superior	**8**	**8**
Totales	**16**	**16**

Cuadro 6.7: Detalle de los errores encontrados en algoritmo propuesto

[1]Reporte en base al algoritmo de análisis de soluciones, presentado en la sección 6.4

6.6.3. Extensión de la metodología a un problema más general

A continuación se muestra un estudio preliminar de la aplicación de esta metodología a un problema más general. Los resultados aquí presentados distan mucho de ser concluyentes. Para el estudio se ha definido un CSP con incertidumbre como sigue:

Definición 6.6.1 (CSP con incertidumbre). Sea $P = (X, D, C)$ un CSP con $X = \{x_1, x_2, ..., x_n\}$, $D = \{D_{x_1}, D_{x_2}, ..., D_{x_n}\}$ y $C = \{c_1, c_2, ..., c_m\}$ restricciones de igualdad sobre las variables. Se dice que P es un *CSP con incertidumbre* si $c_i : f_i(x_1, ..., x_n) = 0 \pm e_i$, con $e_i \geqslant 0\ \forall i = 1..m$.

Considerando la generalidad del problema, se ha realizado una nueva implementación de la metodología, considerando las observaciones realizadas en los capítulos precedentes. El código en lenguaje C++ ha sido incluido en el anexo. Esta nueva implementación utiliza programación orientada a objetos. El conjunto de objetos definidos es mostrado en el cuadro 6.8. En la descripción se ha omitido el objeto *variable*, porque se utilizó el objeto predefinido de la biblioteca de ILOG.

	Nombre	**Descripción**
Objeto	Interval	almacena los límites de un intervalo
Atributo	lower	límite inferior del intervalo
	upper	límite superior del intervalo
Objeto	Box	almacena los límites del dominio para cada variable
Atributo	limits	espacio n-dimensional (vector<Interval>)
Objeto	Constraint	almacena una restricción y su incertidumbre
Atributo	expression	parte izquierda de la ecuación
	uncertainty	incertidumbre en la parte derecha de la ecuación
	allvars	verdadera si todas las variables están en la restricción
	varsinside	variables que forman parte de la restricción
Objeto	Solution	almacena los dominios y los valores solución
Atributo	id	identificador de la solución (único)
	values	valores solución para cada variable (vector<IloNum>)
	limits	dominios de las variables (Box)

Cuadro 6.8: Descripción de los principales objetos definidos

Problema general de una parábola y una recta

Para este ejemplo se ha seleccionado un sistema de ecuaciones con dos incógnitas. La ecuación 6.6.1 describe una parábola en el plano cartesiano, mientras que la ecuación 6.6.2

describe una recta. Ambas ecuaciones poseen incertidumbres asociadas ($e_1 = 0.5$ y $e_2 = 1$, respectivamente).

$$7x^2 - y - 20 = 0 \tag{6.6.1}$$

$$x + 5 - y = 0 \tag{6.6.2}$$

Los dominios para ambas variables han sido fijados en el intervalo [-50, 50].

La figura 6.4 muestra la gráficas del sistema de ecuaciones en torno a los puntos solución. En ella se muestra el cálculo de cajas exteriores para ambos conjuntos solución.

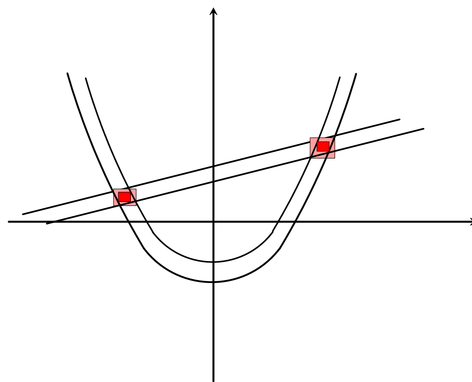


Figura 6.4: Ejemplo de aplicación para una parábola y una recta

Los resultados obtenidos se muestran en el cuadro 6.9.

	x	y
Solución 1	1.9626	6.9626
Caja interna	[1.9534, 1.97168]	[6.9626, 6.9626]
Dominio	[-50, 50]	[5.07143, 50]
Caja externa	[1.9340, 1.9907]	[6.4340, 7.4907]
Solución 2	-1.81974	3.18026
Caja interna	[-1.8295, -1.8099]	[3.18025, 3.18027]
Dominio	[-50, 50]	[-50, 5.07143]
Caja externa	[-1.84786, -1.7912]	[2.68972, 3.67084]

Cuadro 6.9: Resultados del sistema parábola-recta

Para el sistema sin incertidumbre se encontraron dos soluciones: los puntos (1.9626, 6.9626) y (-1.81974, 3.18026). Una vez incluidas las incertidumbres del problema, se calcularon las cajas interiores para cada variable de la solución encontrada. Posteriormente se

aplicó el algoritmo de separación de soluciones, que dividió el espacio correspondiente a la segunda variable en el valor 5.07143. A partir de los nuevos dominios se calcularon las cajas externas, aproximando de esta forma las soluciones del problema.

Problema general con funciones trigonométricas

Este ejemplo muestra la aplicación de la metodología a un problema más general que involucra funciones trigonométricas. La ecuación 6.6.3 describe una curva en el plano cartesiano, formada por la suma de una función logarítmica y una senoidal, mientras que la ecuación 6.6.4 describe una parábola

$$100log(x+10)sin(2x) - y - 50 = 0 \tag{6.6.3}$$

$$2x^2 - y = 0 \tag{6.6.4}$$

Los dominios de las variables fueron restringidos a los intervalos $D_x = [-9, 9]$ y $D_y = [0, 200]$. Por otra parte, se consideraron las incertidumbres $e_1 = 0.3$ y $e_2 = 0.2$, respectivamente. La figura 6.5, muestra la gráfica del sistema de ecuaciones sin considerar las incertidumbres.

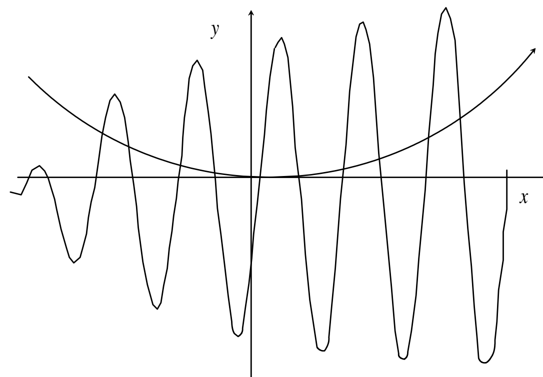


Figura 6.5: Ejemplo de una función senoidal y una parábola

Dado que existen diez soluciones al problema (puntos del plano que cumplen ambas restricciones en forma simultánea), se dividió el espacio de búsqueda inicial en diez subespacios, cada uno conteniendo una sola solución. De esta forma, se calculan las cajas interiores y exteriores utilizando la metodología propuesta.

	x	y
Solución 1	7.54929	113.983
Dominio	[-9, 9]	[99.6985, 200]
Caja interior	[7.54897, 7.54961]	[113.983, 113.983]
Caja exterior	[7.54878, 7.54979]	[113.88, 114.087]
Solución 2	6.53504	85.4135
Dominio	[-9, 9]	[76.3918, 99.6985]
Caja interior	[6.53473, 6.53534]	[85.4135, 85.4135]
Caja exterior	[6.5345, 6.53557]	[85.2995, 85.5274]
Solución 3	4.53856	41.1971
Dominio	[-9, 9]	[31.3682, 46.1494]
Caja interior	[4.53826, 4.53886]	[41.1971, 41.1971]
Caja exterior	[4.53808, 4.53904]	[41.0953, 41.2988]
Solución 4	3.28172	21.5394
Dominio	[0.158495, 9]	[11.6941, 31.3682]
Caja interior	[3.28142, 3.28202]	[21.5394, 21.5394]
Caja exterior	[3.28121, 3.28223]	[21.4326, 21.6461]
Solución 5	1.45862	4.25515
Dominio	[-0.122822, 9]	[2.13945, 11.6941]
Caja interior	[1.45831, 1.45894]	[4.25515, 4.25515]
Caja exterior	[1.4581, 1.45914]	[4.15455, 4.35576]
Solución 6	-5.05478	51.1017
Dominio	[-9, 9]	[46.1494, 59.2359]
Caja interior	[-5.05542, -5.05415]	[51.1017, 51.1017]
Caja exterior	[-5.05595, -5.05362]	[50.9782, 51.2253]
Solución 7	-5.80388	67.3701
Dominio	[-9, 9]	[59.2359, 76.3918]
Caja interior	[-5.8047, -5.80307]	[67.3701, 67.3701]
Caja exterior	[-5.80509, -5.80268]	[67.2645, 67.4757]
Solución 8	0.108977	0.023752
Dominio	[-9, 9]	[0, 2.13945]
Caja interior	[0.108647, 0.109308]	[0.0237439, 0.0237907]
Caja exterior	[0.108594, 0.109528]	[0, 0.123993]
Solución 9	-2.96473	17.5793
Dominio	[-9, 0.158495]	[11.6941, 31.3682]
Caja interior	[-2.96513, -2.96433]	[17.5792, 17.5793]
Caja exterior	[-2.96538, -2.96408]	[17.4777, 17.6808]
Solución 10	-1.70427	5.80904
Dominio	[-9, -0.122822]	[2.13945, 11.6941]
Caja interior	[-1.70464, -1.7039]	[5.80904, 5.80904]
Caja exterior	[-1.70489, -1.70364]	[5.70477, 5.91333]

Cuadro 6.10: Resultados para el sistema seno-parábola

El cuadro 6.10, muestra el resultado de la aplicación del algoritmo general para el problema planteado. Para cada solución, se presentan los valores de la coordenada X e Y, seguido por el subespacio asignado (Nuevo dominio para ambas variables), y los resultados obtenidos para cajas interiores y exteriores. Los valores obtenidos aproximan las soluciones del problema planteado, considerando la influencia de las incertidumbres.

6.7. Conclusión del capítulo

En este capítulo se presentó una metodología para el tratamiento de incertidumbres en sistemas de ecuaciones de distancia y su extensión a problemas más generales. En primer lugar se expuso las propiedades de los CSP de distancia que permiten implementar un algoritmo de clasificación de problemas. Posteriormente se propusieron tres algoritmos: de *separación de puntos*, *separación de soluciones* y *análisis de soluciones*, que integran la metodología propuesta, mejorando las aproximaciones de las soluciones del problema.

Finalmente, se presentaron los resultados preliminares, obtenidos tanto para sistemas de ecuaciones de distancia como para problemas de índole más general. Estos resultados mostraron las debilidades de la metodología actual, y cómo es posible mejorar las soluciones con la nueva propuesta.

Capítulo 7

Conclusiones

En este trabajo se estudió una metodología para el tratamiento de incertidumbre en CSP numéricos, y sus aplicaciones en sistemas de ecuaciones de distancia. Se desarrolló además, un estudio teórico y práctico de las técnicas modernas de la programación con restricciones adaptadas a dominios continuos, y sus aplicaciones en problemas reales.

Desde el punto de vista teórico, se propusieron importantes cambios a los algoritmos originales que mejoran la aproximación de soluciones desde el punto de vista de la correctitud. Se demostró que la modificación del algoritmo de separación de puntos reduce los problemas causados por el corte inapropiado del espacio de búsqueda, aumentando los casos para los cuales la metodología permite resolver el problema satisfactoriamente. Por otro lado, la inclusión de un algoritmo de análisis posterior de resultados mostró su efectividad en la detección de problemas de corte y cálculo de cajas exteriores, permitiendo una mejor interpretación de los resultados. Este punto adquiere gran importancia, ya que la metodología original asume la separabilidad de los puntos solución, y no es capaz de detectar fallos en la determinación de las soluciones debidos a violaciones de las premisas.

Otro punto importante es la posibilidad teórica de determinar, de una forma cualitativa, el tipo de configuración solución que poseerá el sistema, a través del estudio de las observaciones propuestas en la sección 5.1. Si bien no se profundizó en la generalización de estas observaciones a mayores dimensiones, es posible concluir que un estudio más detallado podría incrementar fuertemente la potencialidades de este enfoque. Además, deja en claro que no es recomendable aplicar esta metodología sin poseer conocimiento previos del problema particular, ya que los resultados poseen directa relación con el nivel de incertidumbre y las relaciones particulares de los puntos.

La ventaja de este enfoque por sobre otras metodologías, como la propuesta en [KB99] o en [Bac98], es que aproxima las soluciones de forma interna y externa. En otras palabras, además de entregar una aproximación que encierra todo el conjunto solución, determina un espacio de tolerancia (incluido en la aproximación interior), que sólo posee puntos válidos para el problema. Esto es importante porque en general las técnicas de consistencia no aseguran la presencia de solución, sino sólo la ausencia de ella en un espacio determinado. Por esta razón, una doble aproximación (i-consistente y e-consistente), entrega una aproximación más precisa para el problema.

Desde el punto de vista práctico, se implementó un nuevo algoritmo basado en la metodología original, y que integra las nuevas propuestas en relación con los algoritmos de separación y análisis posterior de soluciones, que fue comparado con el algoritmo original, obteniendo aproximaciones más precisas (y correctas) para un conjunto de casos de prueba.

Por otra parte, se implementó un algoritmo que extiende esta metodología de tratamiento de incertidumbres, permitiendo su utilización en CSP numéricos de índole más general. Si bien, esta propuesta posee, en principio, grandes limitaciones respecto a los algoritmos de separación (ya que no es posible asegurar simetrías) y en la determinación cualitativa de las soluciones, permite una aproximación inicial para las soluciones de aquellos sistemas que, por sus características, no pueden omitir la influencia de las incertidumbres en el cálculo de sus soluciones.

Propuestas de Trabajos Futuros

Varios son los trabajos que pueden desarrollarse a partir de este estudio. En particular, la generalización de las observaciones propuestas para un algoritmo de análisis previo, que proporcione una clasificación de problemas en *abordables a través de esta metodología* y aquellos que, a priori, se sabe que entregarán soluciones incorrectas o imprecisas.

Por otro lado, en el cálculo de cajas interiores, sólo se asegura la obtención de una cota inferior, pero en ningún momento se tiene en cuenta la optimización de esta cota, de tal forma que sea la *mayor caja interior inscrita en el conjunto solución*. Esto se debe al procedimiento utilizado para calcular la cota inferior, que extiende los dominios de cada coordenada hasta el límite de su capacidad, restando posibilidades a todas aquellas coordenadas calculadas después de cada extensión. Este problema podría llegar a ser tan complicado, como el determinar el conjunto solución exacto.

Para finalizar, se observó un aumento en el tiempo de resolución requerido por el algoritmo ante iguales problemas. Este hecho puede deberse al tamaño del espacio de búsqueda resultante después de aplicar la estructura de árbol para la separación de soluciones. Debido a que los subespacios en el nuevo algoritmo son, en general, mayores que en el algoritmo original, las técnicas de consistencia requieren mayores tiempos para realizar el filtraje necesario.

Parte IV

Apéndices

Apéndice A

ILOG Concert/Solver

A.1. Introducción

Este apartado pretende ser una ayuda para trabajar con las bibliotecas ILOG Solver en la resolución de problemas de satisfacción de restricciones. Las bibliotecas para C++ de ILOG Solver permiten realizar definiciones de alto nivel y resolver CSP, manteniendo un cierto control sobre las estructuras de datos. ILOG Solver trabaja en conjunto con la tecnología ILOG Concert, que proporciona clases para definir el *modelo*, mientras que ILOG Solver proporciona aquellas necesarias para resolverlo.

Para comenzar es necesario definir el entorno (variables, restricciones, etc.), que contendrá el problema en sí. Posteriormente, se aplican las bibliotecas de resolución. Todas las clases *Concert* comienzan por **Ilo**, mientras la clases *Solver* comienzan por **Ilo** o **Ilc**.

En el caso de modelos de programación lineal es posible utilizar la herramienta integrada CPLEX, que permite resolver este tipo de modelos en lugar de utilizar Solver. Ambos utilizan la misma definición formulada con Concert.

A.2. Definición de un modelo (ILOG Concert)

Existe un conjunto de convenciones que es importante tener en cuenta antes de comenzar la definición de modelos. Las principales son:

1. Los nombres de las clases se han formado concatenando palabras, con la primera letra de cada palabra en mayúscula. Por ejemplo: $IloIntVar$.

2. Para nombres de argumentos, instancias y funciones miembro se utiliza al comienzo una letra minúscula, y en caso de poseer más de una palabra se coloca la primera letra de la palabra en mayúscula. Por ejemplo: $IloAnyVar :: setDomain$.

3. Los nombres de los datos miembros comienzan con underscore, seguido del nombre. Por ejemplo: $char * _boys;$.

4. Generalmente los accesos comienzan con la palabra *get* (como $getValue()$), a menos que sean miembros booleanos, en cuyo caso se utiliza la palabra *is* (como en $isBig()$). Para establecer valores se usa frecuentemente la palabra *set* (como en $setConstraints()$).

5. Además, se utilizan los siguientes nombres para tipos de datos básicos:

   - **IloInt** para enteros largos con signo.
   - **IloAny** para punteros (equivale a **void***).
   - **IloNum** para valores de punto flotante de doble precisión.
   - **IloBool** para valores booleanos (**IloTrue** y **IloFalse**).

6. Por último, los paréntesis cuadrados denotan dominios de las variables. Por ejemplo, [0 2] significa que la variable puede tomar el valor 0 o el valor 2. Por el contrario la expresión [0 .. 2] denota el rango de valores (intervalo de enteros) entre 0 y 2, o sea 0, 1 y 2.

Un *entorno* es una instancia de la clase *IloEnv*, y es el encargando de administrar el modelo interno. Además, manipula las salidas, memoria y la finalización de los algoritmos entre otras cosas. Un *modelo* es una instancia de la clase $IloModel$, y contiene los objetos del modelo, como las variables o las restricciones. Es creado utilizando el entorno.

Para crear un modelo, el primero paso es definir una instancia de la clase *IloEnv* con la siguiente instrucción:

```
IloEnv entorno;
```

Una vez creado el entorno, se procede a la creación de los objetos del modelo (variables y restricciones).

```
IloNumVar x(entorno, 0, 10, ILOINT);  //Variable x entera, dominio 0-10
IloConstraint ct = (x != 0);          //Restricción ct unaria
```

Luego se crea el modelo y se llena con los objetos:

```
IloModel modelo(entorno);      //Creacion del modelo
modelo.add(ct);                //Ingreso de un restricción
```

Una vez que se hay resuelto el problema, es posible solicitar la memoria de todos los objetos y limpiar las estructuras internas del entorno llamando al método *IloEnv* :: *end*. Es recomendable hacer esto antes de salir de la aplicación.

A.3. Resolución del modelo (ILOG Solver)

Una vez creado el modelo, se crea una instancia de la clase *IloSolver* para proceder a la resolución. Las operaciones básicas son *extraer el modelo* usando el método *IloSolver* :: *extract* y luego *resolver el modelo* usando el método *IloSolver* :: *solve*. A continuación se muestra un ejemplo:

```
IloSolver solver(entorno);  //Creacion de la instancia
solver.extract(modelo);     //Extracción del modelo
```

También existe una forma resumida de realizar las operaciones anteriores:

```
IloSolver solver(modelo);   //Creacion de la instancia y extracción
```

Durante la extracción, se escanean todos los elementos del modelo y se crean objetos de ellos. Estos objetos estan usualmente precedidos del prefijo *Ilc*. Se puede acceder a ellos usando los métodos del Solver. Por ejemplo:

```
IlcIntVar solx = solver.getIntVar(x); //Valor de la variable x
```

A continuación se muestra un ejemplo completo del código para resolver un CSP. El problema posee una variable (x) entera, con dominio [0 .. 10] y una restricción ($x \neq 0$). El resultado del programa se muestra en pantalla (a través del metodo *IloSolver* :: *out*()) en corchetes ([1]).

```
#include <ilsolver/ilosolver.h>

int main(){
  IloEnv entorno;
  IloNumVar x(entorno, 0, 10, ILOINT);
  IloConstraint ct = (x != 0);
  IloModel modelo(entorno);
  modelo.add(ct);
```

```
  IloSolver solver(modelo);
  solver.solve();
  solver.out()<<solver.getIntVar(x)<<endl;

  entorno.end();
}
```

Existen además dos sectores de memoria distintos que pueden ser utilizados para almacenar objetos deseados. Estos sectores son automáticamente liberados por los destructores de cada clase. En el caso del entorno, la instrucción para almacenar objetos en memoria es:

```
MiObjeto* miobjeto = new (entorno) MiObjeto();
```

Y para el solver:

```
MiObjeto* miobjeto = new (solver.getHeap()) MiObjeto();
```

La memoria, en el caso del solver es liberada cuando se realiza una llamada a $IloSolver :: end()$, o cuando el entorno recibe una llamada $IloEnv :: end()$.

A.4. Elementos de un modelo

Hasta el momentos se ha analizado la forma de construir y resolver un modelo en particular. A continuación se entregan elementos con los cuales es posible definir un modelo.

A.4.1. Tipos de Variables

Existen básicamente dos tipos de variables que pueden ser declaradas de diferente forma dependiendo de los usos que se pretenda dar. La primera pertenece a la clase $IloNumVar$ utilizada para valores de punto flotante o enteros. La segunda pertenece a la clase $IloNumVarArray$ y corresponde a un conjunto de variables (arreglo). Algunos ejemplos son:

```
IloNumVar x(entorno, 0, 10, ILOINT);   //Variable x entera
IloNumVar y(entorno, 4, 9, ILOINT);    //Variable y entera
IloNumVar z(entorno, -3, 6, ILOFLOAT); //Variable z flotante

IloNumVarArray a(entorno, 10, 2, 8, ILOINT); //Arreglo a (10 vars)
IloNumVarArray b(entorno, 2, x, y);          //Arreglo b (2 vars)
IloNumVarArray c(entorno, 10, 1, 9);         //Arreglo c (10 vars)
```

En el ejemplo, las variables x e y son variables enteras con dominios [0 .. 10] y [4 .. 9] respectivamente, mientras que la variable z es una variable continua (punto flotante de doble

precisión) y su dominio es el intervalo [-3 , 6]. Por otro lado, la variable a es en realidad un conjunto de 10 variables enteras con dominio [2 .. 8], mientras que b es un conjunto de 2 variables (x e y) y c un arreglo de 10 variables entre 1 y 9.

A.4.2. Expresiones

Las expresiones se forman por la combinación de operadores, funciones, otras expresiones, variables, etc. Los operadores aritméticos básicos ($+$, $-$, $*$ y $/$) han sido sobrecargados para trabajar con los objetos de las clases (variables, numeros, etc.). Existen además funciones algebraicas de uso común, por ejemplo:

- **IloAbs(x):** Valor absoluto del número entre paréntesis.
- **IloSquare(x):** Representa el cuadrado del valor (x^2).
- **IloExponent(x):** Representa el valor e^x.
- **IloLog(x):** Logaritmo natural de x.
- **IloMax(x,y):** Valor máximo entre x e y.
- **IloMin(x,y):** Valor mínimo entre x e y.
- **IloPower(x,y):** Representa el valor x^y.

También existen funciones que permiten trabajar con conjuntos de objetos, por ejemplo, un arreglo de variables. Las más comunes son la sumatoria de los elementos del arreglo y el producto escalar entre dos vectores.

- **IloSum(a):** Sumatoria de los valores del arreglo a.
- **IloScalProd(a,b):** Producto escalar de a por b.

En el ejemplo anterior se asume que tanto a como b son vectores, pero estos pueden estar formados tanto por variables como por valores fijos (arreglos de la clase $IloNumArray$).

A.4.3. Tipos de Restricciones

Los tipos de restricciones varían según sean las variables a quienes se apliquen. Las más comunes son *igualdades*, *desigualdades* y *diferencias*. Los operadores clásicos de C++ (==, <, <=, >=, >, ! =) han sido sobrecargados para este efecto. Este tipo de restricciones son más bien "clasicas".

Existen además restricciones denominadas *simbólicas* [ILO00b], que permiten definiciones especiales y frecuentemente utilizadas por los modelos. Por ejemplo, es muy frecuente la restricción *"todas las variables deben ser diferentes"*, por lo que existe una manera rápida de realizar esto a partir de un arreglo de variables con la expresión $IloAllDiff()$. Otra restricción predefinida de gran utilidad es $IloDistribute(c, vl, vr)$. Ésta puede ser usada para, por ejemplo, contar el número de variables que toman un valor dado en un arreglo.

La declaración de restricciones se puede incluir directamente al modelo, o se puede definir previamente. Por ejemplo:

```
IloConstraint r1 = (x + 1 == y);   //Define la restricción r1.
IloConstraint r2 = IloAllDiff(a);  //Define la restricción r2.

modelo.add(r1);                    //Incluye restricción definida
modelo.add(x != y);                //Incluye una nueva restricción
```

La primera línea define una restricción de diferencia entre x e y llamada $r1$. Esta no forma parte del modelo, a menos que se incluya explícitamente. La segunda línea define la restricción *todas la variables del arreglo* **a** *deben ser diferentes*. La cuarta línea incluye una restricción (declarada previamente) al modelo. Por el contrario en la quinta línea una restricción es incluida directamente en el modelo sin definición previa (al contrario de la línea 9). Un detalle importante es que el modelo así propuesto tendía dos variables y dos restricciones (la restricción de diferencias mostrada en la línea 2 nunca se incluyó en el modelo).

En el ejemplo anterior se mostró el uso de $IloAllDiff()$, una restricción simbólica que se interpreta como *cada valor del arreglo entre paréntesis debe ser distinto a los demás*. A diferencia de las restricciones explícitas, este tipo de restricciones evitan la escritura de un conjunto (potencialmente) exponencial de restricciones en el modelo. Otro ejemplo de este tipo, con la restricción $IloDistribute()$ se muestra a continuación:

```
IloNumVarArray ocurrencias(entorno, 5, 0, 4, ILOINT);
IloNumArray valores(entorno, 5);
IloNumVarArray variables(entorno, 5, a, b, c, d, e);
```

```
modelo.add(IloDistribute(entorno, ocurrencias, valores, variables));
```

La primera línea define un arreglo (*ocurrencias*) de 5 variables con dominio entre [0 .. 4]. La segunda define un arreglo de 5 valores (que deben ser llenados a través del operador [ ]. La tercera línea define un arreglo (*variables*) de 5 variables (a, b, c, d, e). La restricción de la línea 5 es incluida directamente en el modelo, y su significado es *cada valor* **valores[i]** *debe aparecer* **ocurrencias[i]** *veces en las variables del conjunto* **variables**.

Una característica adicional e importante de destacar es el echo de que *Solver* interpreta los valores $verdadero$ y $falso$ como 1 y 0 respectivamente. Esto es útil cuando se desea definir $meta - restricciones$, o sea, restricciones que involucran otras restricciones. A continuación se entrega un ejemplo de esto:

```
IloNumVar bin = (x < 4);              //Definicion variable bin
modelo.add(bin + (y > 2) == 1);       //Meta restricción
```

En el ejemplo, la variable *bin* puede tomar los valores 1 o 0, ya se que se cumpla o no la restricción $x < 4$. Por otra parte, la restricción incluida en el modelo es en realidad una meta-restricción, puesto que obliga a que se cumpla $x < 4$ ó $y > 2$, pero una y sólo una de ellas.

A.4.4. Función Objetivo

Una de las característica más frecuentes de los problemas que involucran restricciones es que poseen un cierto objetivo que se desea optimizar. Por ejemplo, minimizando alguna variable definida en función de otras o maximizando una expresión dada. *Solver* proporciona una clase (*IloObjective*) que permite definir funciones objetivo. Ejemplo:

```
IloObjective obj = IloMaximize(entorno, x);
modelo.add(obj);

modelo.add(IloMinimize(entorno,IloSum(a) + x * x));
```

En el primer ejemplo (líneas 1 y 2) se define una instancia de la clase *IloObjective* (*obj*) como la maximización de una variable (x). Luego se incluye el objetivo en el modelo, tal como se incluían las restricciones. En el segundo ejemplo el objetivo es pasado directamente al modelo, como la minimización de la suma de un vector (a) más el cuadrado de la variable x (x^2).

A.5. Manipulación de la resolución

Cuando se resuelve un CSP existen diferentes posibilidades de selección de variables, dominios, etc, de acuerdo a algún objetivo dado. *IloSolver* posee clases (*IloGoal* e *IloGenerate*) que permiten manipular estos criterios. Por ejemplo:

```
IloEnv entorno; IloModel modelo(entorno);

IloNumVar x(entorno, 0, 2, ILOINT), y(entorno, 0, 2, ILOINT);
modelo.add(x!=y); IloSolver solver(modelo);

IloGoal objetivo=IloGenerate(entorno, IloNumVarArray(entorno, 2, x, y));

if (solver.solve(objetivo)){
    solver.out()<<"x: "<<solver.getValue(x);
    solver.out()<<"y: "<<solver.getValue(y)<<endl;
} else
    solver.out()<<"Sin solucion"<<endl;
```

Este ejemplo muestra la utilización de *IloGenerate* para establecer un orden de selección en la búsqueda. En este caso la variable x será instanciada antes que la variable y, por lo que la primera solución encontrada será $x = 0$ e $y = 1$. Por otro lado, si se desea encontrar todas las soluciones, es posible definir una instancia de la clase *IlcSearch* y asignar una nueva búsqueda de la siguiente forma:

```
IloGoal objetivo=IloGenerate(entorno, IloNumVarArray(entorno, 2, x, y));
IlcSearch busqueda = solver.newSearch(objetivo);

while (busqueda.next()){
    solver.out()<<"x: "<<solver.getValue(x);
    solver.out()<<"y: "<<solver.getValue(y)<<endl;
}
```

El ejemplo anterior generará todas las soluciones, siempre instanciando la variable x antes de la variable y. En el caso de *IloGenerate*, es posible entregar como argumento criterios más específicos, como *instanciar primero la variable con menor dominio* o *instanciar primero la variable con menor o mayor límite*, por ejemplo. A continuación un listado de las opciones más comunes.

- **IlcChooseMinSizeInt:** Comenzar por variable con menor dominio.
- **IlcChooseMaxSizeInt:** Comenzar por variable con mayor dominio.
- **IlcChooseMinMinInt:** Variable con menor valor mínimo.

- **IlcChooseMaxMinInt:** Variable con mayor valor mínimo.
- **IlcChooseMinMaxInt:** Variable con menor valor máximo.
- **IlcChooseMaxMaxInt:** Variable con mayor valor máximo.

El criterio es agregado al final de la declaración *IloGenerate*:

```
IloGenerate(entorno, a, IlcChooseMaxSizeInt);
```

Es posible limitar la búsqueda de solución, entregando criterios de *tiempo límite* o el *Número de selecciones*. Por ejemplo:

```
solver.setTimeLimit (100);
solver.setFailLimit (1000);
solver.setOrLimit (2000);
```

Para desplegar información referente al tiempo, número de fallos, puntos de selección, etc., existe el método *printInformation*() de la clase *IloSolver*. Una salida típica de este método se muestra a continuación:

```
Number of fails              : 0
Number of choice points      : 2
Number of variables          : 3
Number of constraints        : 2
Reversible stack (bytes)     : 4044
Solver heap (bytes)          : 4044
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 9136
Total memory used (bytes)    : 33400
Running time since creation  : 0.01
```

Por último, en el caso de problemas de programación lineal, es posible utilizar las bibliotecas de *Cplex* con el modelo *Concert* para resolverlo. A continuación se presenta un ejemplo.

```
IloCplex lineal(modelo);
lineal.solve();
```

Y obtener el valor de una variable (o de la función objetivo) con:

```
lineal.getValue(variable);
lineal.getObjValue(variable);
```

A.6. Ejemplo

Un ejemplo simple de la utilización de esta herramienta es el siguiente: *Se tiene una cantidad de dinero a devolver, y monedas de varias denominaciones. Se desea encontrar el mínimo número de monedas necesarias para entregar la cantidad de dinero solicitada.*

```
#include <ilsolver/ilosolver.h>

int main(int argc, char** argv){
  IloEnv entorno;
  if (argc<3) {
    cerr<<argv[0]<<" <vuelto> <tipo de monedas> <val1> ... <valn>"<<endl;
  }else{
    try {
      IloInt vuelto = atoi(argv[1]);    //Cantidad a devolver
      IloInt nmon = atoi(argv[2]);      //Cantidad de denominaciones

      IloNumArray vmon(entorno, nmon);  //Denominación de las monedas
      IloNumVarArray cmon(entorno, nmon, 0, vuelto, ILOINT); //Variables

      IloModel modelo(entorno);         //Declaración del modelo

      for (IloInt i=0;i<nmon;i++)       //Lectura de la denominación
        vmon[i]=atoi(argv[3+i]);        //desde la entrada

      modelo.add(IloScalProd(vmon, cmon)==vuelto);     //Restricción
      modelo.add(IloMinimize(entorno, IloSum(cmon)));

      IloSolver solver(modelo);         //Declaración de una instancia
      solver.solve();                   //Resolución

      for (IloInt i=0;i<nmon;i++)       //Despliegue resultados
        solver.out()<<solver.getValue(cmon[i])<<" de "<<vmon[i]<<endl;
      solver.out()<<endl<<endl;
      solver.printInformation();
    }
    catch (IloException& ex) {
      cerr<<"Error: "<<ex<<endl;
    }
  }
  entorno.end();
  return 0;
}
```

A.7. Consideraciones sobre dominios continuos

Cuando se trabaja en dominios continuos las necesidades suelen ser un poco diferentes. Por ejemplo, en ocasiones interesa un dominio de soluciones o simplemente una cierta precisión en el cálculo de valores óptimos. Una de las funciones utilizada para este efecto es

IloSplit. A continuación un conjunto de métodos útiles (de la clase *IloSolver*) para el trabajo sobre variables reales.

- **setDefaultPrecision(p)** Configura la precisión a utilizar como p.
- **useNonLinConstraint(_)** Restricciones no lineales (IlcTrue/ IlcFalse).

Las estrategias de división de dominio y propagación pueden especificarse como parámetros de *IloSplit* como en el siguiente ejemplo:

```
IloSplit(entorno, variables, IloTrue, p) //3B o Bound con precisión p
IloSplit(entorno, variables)             //2B o Box
```

Además existen algunas funciones adicionales:

```
IloDichotomize()     //bisección sobre una sola variable
IloGenerateBounds()  //filtraje por 3B o Bound sin bisección
```

A continuación se entrega un ejemplo de código para resolver un problema de 2 ecuaciones con 2 incógnitas (creado por Jean-Pierre [Nev02a]).

```
#include <ilsolver/ilosolver.h>

void main(){
  IloEnv env;

  IloNumVar x(env, -10, 10);  //Variable x continua, dominio [-10,10]
  IloNumVar y(env, -20, 20);  //Variable y continua, dominio [-20,20]
  IloNumVarArray vars(env, 2, x, y);  //Arreglo de variables x e y

  IloModel model(env);        //Creacion del modelo
  model.add(x*IloCos(y) + IloSquare(x)*IloCos(y) == 1);
  model.add(IloSquare(x) + IloSquare(y) == 1);

  IloSolver solver(env);             //La resolución
  solver.setDefaultPrecision(1e-10); //Precision (10 dígitos)

  //Uso de funciones numéricas (Box?, Newton?)
  //solver.useNonLinConstraint(IlcFalse);

  solver.extract(model);            //Extracción del modelo

  //Uso de propagación mas fuerte (3B, Bound y su precisión)
  //IlcSearch search=solver.newSearch(IloSplit(env,vars,IloTrue,1e-6));

  //Resolución simple sin llamar a propagaciones mas fuertes
  IlcSearch search=solver.newSearch(IloSplit(env,vars));

  while (search.next()){
```

```
    cout<<"Solución "<<" x ["<<solver.getMin(x)<<", "
       <<solver.getMax(x)<<"]    y ["<<solver.getMin(y)<< ", "
       <<solver.getMax(y)<<"]"<<endl<<endl;
  }
  solver.printInformation();
  env.end();
}
```

El ejemplo resuelve un sistema de ecuaciones entregando las dos posibles soluciones. Las instrucciones comentadas (línea 18 por ejemplo), son ejemplos de aplicación de los criterios para el filtraje de dominios.

Apéndice B

Algoritmo General

A continuación se presenta la implementación del algoritmos de tratamiento de incertidumbres en CSP numéricos. Para ello se utilizó como base el lenguaje de programación orientado a objetos C++ y el conjunto de bibliotecas provistas por ILOG Solver para resolución de los sistemas y la aplicación de técnicas de consistencia.

```
#include<ilsolver/ilosolver.h>
#include<vector>
using namespace std ;

ILOSTLBEGIN

// Class "Interval", storage interval's limits for variables or uncertainty
class Interval{
  IloNum lower;                      // Minimum value of interval
  IloNum upper;                      // Maximum value of interval

public :
  Interval(){;}

  Interval(IloNum inf, IloNum sup){
    lower = inf;
    upper = sup;
  }

  IloNum getLb(){
    return lower;
  }

  IloNum getUb(){
    return upper;
  }


  void setLb(IloNum inf){
    lower = inf;
```

```
  }

  void setUb(IloNum sup){
    upper = sup;
  }

  friend ostream& operator<<(ostream& os,const Interval &i){
    os<<"["<<i.lower<<"; "<<i.upper<<"]";
    return os;
  }
};


// Class "Box", storage domain's limits for each variable
class Box{
  vector<Interval> limits;          // Space "var" dimensional

public:
  Box(){;}

  Box(IloInt number){
    limits.resize(number);
  }

  void addInterval(Interval i){
    limits.push_back(i);
  }

  void setVarInterval(IloInt pos, Interval i){
    limits[pos] = i;
  }

  Interval getVarInterval(IloInt pos){
    return limits[pos];
  }

  void setDimension(IloInt number){
    limits.resize(number);
  }

  void clear(){
    limits.clear();
  }
};
```

```
// Class "Constraint", storage constraint and uncertaninty
class Constraint{
  IloExpr expression;               // Constraint's expression
  Interval uncertainty;             // Uncertainty in right part
  IloBool allvars;                  // True: all variables inside
  IloNumArray varsinside;           // Variables inside expression

public :
  Constraint(){;}

  Constraint(IloExpr expr, IloNum inf, IloNum sup){
    expression = expr;
    uncertainty.setLb(inf);
    uncertainty.setUb(sup);
    allvars = true;
  }

  void setExpresion(IloExpr expr){
    expression = expr;
  }

  void setUncertainty(IloNum inf, IloNum sup){
    uncertainty.setLb(inf);
    uncertainty.setUb(inf);
  }

  IloExpr getExpr(){
    return expression;
  }

  IloNum getLb(){
    return uncertainty.getLb();
  }

  IloNum getUb(){
    return uncertainty.getUb();
  }

  void setAllVarInside(IloBool a){
    allvars = a;
  }

  void setVarInConstraint(IloNumArray var){
    varsinside = var;
  }

  IloBool isAllVarInside(){
    return allvars;
  }

  IloNumArray getVarInConstraint(){
    return varsinside;
  }
};
```

```
// Class "Solution", hold solutions values and variables' domains
class Solution{
  IloInt id;                          // Solutions identifier
  vector<IloNum> values;              // Values for all variables
  Box limits;                         // Variables' domains

public:
  Solution(){;}

  Solution(IloInt i, IloInt nvars){
    id = i;
    values.resize(nvars);
  }

  IloInt setId(IloInt i){
    id = i;
  }

  void setBox(Box domains){
    limits = domains;
  }

  void setVarValue(IloInt varid, IloNum value){
    values[varid] = value;
  }

  void setVarLb(IloInt varid, IloNum lb){
    Interval aux = limits.getVarInterval(varid);
    aux.setLb(lb);
    limits.setVarInterval(varid, aux);
  }

  void setVarUb(IloInt varid, IloNum ub){
    Interval aux = limits.getVarInterval(varid);
    aux.setUb(ub);
    limits.setVarInterval(varid, aux);
  }

  IloNum getVarValue(IloInt varid){
    return values[varid];
  }

  Interval getVarInterval(IloInt varid){
    return limits.getVarInterval(varid);
  }

  IloInt getId(){
    return id;
  }
```

```
  bool operator==(Solution &s2){
    for (IloInt i=0;i<values.size();i++)
      if (values[i]-s2.values[i] > 1e-4)
         return false;
    return true;
  }
};


// Class "Problem" hold the object of problem (variables, ..
// ..constraints, solutions points, etc.)
class Problem{
  vector<Constraint> constraints;   // Constraints of problem
  vector<Solution> solutions;       // Solutions of problem
  vector<Box> i_box;                // intern-Box solutions
  vector<Box> e_box;                // extern-box solutions
  IloNumVarArray variables;         // Variables of problem
  Box initdomain;                   // Initial variables' domains
  IloInt nvars;                     // Number of variables
  IloInt ncons;                     // Number of constraints
  IloInt nsols;                     // Number of solutions
  IloNum prec;                      // Precition of calculs

  vector<IloConstraint> sconst;     // Special Constraints for solving

public:
  Problem(){
    nvars = 0;
    ncons = 0;
    nsols = 0;
    prec = 1e-8;
  }

  void addVars(IloNumVarArray v){
    variables = v;
    nvars = v.getSize();
    Box aux(nvars);
    for (IloInt i=0;i<nvars;i++)
      aux.setVarInterval(i, Interval(v[i].getLb(),v[i].getUb()));
    initdomain = aux;
  }

  void addConstraint(IloExpr e, IloNum lb, IloNum ub){
    Constraint aux(e,lb,ub);
    constraints.push_back(aux);
    ncons++;
    if ((prec > IloAbs(ub - lb)/4) && IloAbs(ub - lb)>0)
      prec = IloAbs(ub - lb)/4;
  }

  void setPrecision(IloNum p){
    prec = p;
  }
```

```
void addConstraint(IloExpr e, IloNum val){
  Constraint aux(e,-val/2,val/2);
  constraints.push_back(aux);
  ncons++;
  if ((prec > val/4) && val > 0)
    prec = val/4;
}

void addConstraint(IloExpr e){
  Constraint aux(e,0,0);
  constraints.push_back(aux);
  ncons++;
}

void setVarInConstraint(IloNum pos, IloNumArray var){
  constraints[pos].setAllVarInside(false);
  constraints[pos].setVarInConstraint(var);
}

void sortSolsByCoord(IloInt cord){
  if (cord == -1){
    for(IloInt i=0;i<nsols-1;i++)
      for(IloInt j=i+1;j<nsols;j++)
        if (solutions[i].getId()>solutions[j].getId()){
          Solution aux = solutions[i];
          solutions[i] = solutions[j];
          solutions[j] = aux;
        }
  } else {
    for(IloInt i=0;i<nsols-1;i++)
      for(IloInt j=i+1;j<nsols;j++)
        if (solutions[i].getVarValue(cord)>solutions[j].getVarValue(cord)){
          Solution aux = solutions[i];
          solutions[i] = solutions[j];
          solutions[j] = aux;
        }
  }
}

void splitDomain(vector<vector<IloInt> > ds){
  IloInt sols = ds[0].size();
  if (sols>1){
    IloNum distance = 0.0;
    IloInt coordenate;
    IloNum point = 0.0;
    for (IloInt i=0;i<nvars;i++)
      for (IloInt j=1;j<sols;j++){
        IloNum diff = solutions[ds[i][j]].getVarValue(i) -
          solutions[ds[i][j-1]].getVarValue(i);
        if (distance < diff){
          distance = diff;
          coordenate = i;
          point = solutions[ds[i][j-1]].getVarValue(i) + diff/2;
        }
```

```
      }

    for (IloInt j=0; j<sols; j++)
      if (solutions[ds[coordenate][j]].getVarValue(coordenate)<point){
        solutions[ds[coordenate][j]].setVarUb(coordenate,point);
      } else {
        solutions[ds[coordenate][j]].setVarLb(coordenate,point);
      }

    vector<vector<IloInt> > dssup;
    dssup.resize(nvars);

    for (IloInt i=0;i<nvars;i++)
      for (IloInt j=sols-1;j>=0;j--)
        if (solutions[ds[i][j]].getVarValue(coordenate)>point){
          dssup[i].insert(dssup[i].begin(),ds[i][j]);
          ds[i].erase(ds[i].begin() + j);
        }

    splitDomain(ds);
    splitDomain(dssup);
  }
}

void separateSols(){
  if (nsols>1){
    vector<vector<IloInt> > ds;
    for (IloInt i=0;i<nvars;i++){
      sortSolsByCoord(i);
      vector<IloInt> aux;
      for (IloInt j=0;j<nsols;j++)
        aux.push_back(solutions[j].getId());
      ds.push_back(aux);
    }
    sortSolsByCoord(-1);
    splitDomain(ds);
  }
}

bool isThisVarInConstraint(IloInt var, IloInt con){
  if (constraints[con].isAllVarInside()){
    return true;
  } else {
    IloNumArray aux = constraints[con].getVarInConstraint();
    if (aux[var])
      return true;
    return false;
  }
}
```

```
void extendRightDomain(IloEnv *env,IloInt sol,IloInt var){
  try {
    IloNum maxextend = initdomain.getVarInterval(var).getUb();
    for (IloInt k=0;k<ncons;k++)
      if (isThisVarInConstraint(var,k)){
        IloModel model(*env);

        for (IloInt i=0; i<nvars; i++){
          Interval aux = i_box[sol].getVarInterval(i);
          variables[i].setBounds(aux.getLb(),aux.getUb());
          model.add(variables[i]);
        }

        IloExpr exp = constraints[k].getExpr();
        IloNum inf = constraints[k].getLb();
        IloNum sup = constraints[k].getUb();
        model.add(exp <= inf || exp >= sup);

        IloNum infinity = initdomain.getVarInterval(var).getUb();
        variables[var].setBounds(variables[var].getUb(),infinity);

        IloSolver solver(model);
        if (solver.solve(IloGenerateBounds(*env,variables,prec)))
          if (solver.getMin(variables[var])<maxextend)
            maxextend = solver.getMin(variables[var]);

        solver.end();
        model.end();
      }

    Interval aux = i_box[sol].getVarInterval(var);
    aux.setUb(maxextend);

    i_box[sol].setVarInterval(var,aux);
  }
  catch (IloException& ex){
    cerr<<"Error in Problem::extendRightDomain(): "<<ex<<endl;
    end();
    exit(1);
  }
}

void extendLeftDomain(IloEnv *env,IloInt sol,IloInt var){
  try {
    IloNum maxextend = initdomain.getVarInterval(var).getLb();
    for (IloInt k=0;k<ncons;k++)
      if (isThisVarInConstraint(var,k)){
        IloModel model(*env);
        for (IloInt i=0; i<nvars; i++){
          Interval aux = i_box[sol].getVarInterval(i);
          variables[i].setBounds(aux.getLb(),aux.getUb());
          model.add(variables[i]);
        }
```

```
        IloExpr exp = constraints[k].getExpr();
        IloNum inf = constraints[k].getLb();
        IloNum sup = constraints[k].getUb();
        model.add(exp <= inf || exp >= sup);

        IloNum minusinfinity = initdomain.getVarInterval(var).getLb();
        variables[var].setBounds(minusinfinity, variables[var].getLb());

        IloSolver solver(*env);
        solver.useNonLinConstraint(); /* Optional */
        solver.extract(model);

        if (solver.solve(IloGenerateBounds(*env,variables,prec)))
          if (solver.getMax(variables[var])>maxextend)
            maxextend = solver.getMax(variables[var]);
        solver.end();
        model.end();
      }

    Interval aux = i_box[sol].getVarInterval(var);
    aux.setLb(maxextend);

    i_box[sol].setVarInterval(var,aux);
  }
  catch (IloException& ex){
    cerr<<"Error in Problem::extendLeftDomain(): "<<ex<<endl;
    end();
    exit(1);
  }
}

void eBox(IloEnv *env){
  try {
    for (IloInt k=0; k<nsols; k++){
      IloModel model(*env);
      for (IloInt i=0; i<nvars; i++){
        Interval aux = solutions[k].getVarInterval(i);
        variables[i].setBounds(aux.getLb(),aux.getUb());
      }

      for (IloInt i=0; i<ncons; i++){
        IloExpr exp = constraints[i].getExpr();
        IloNum inf = constraints[i].getLb();
        IloNum sup = constraints[i].getUb();
        model.add(exp >= inf);
        model.add(exp <= sup);
      }
      IloSolver solver(*env);
      solver.useNonLinConstraint(); /* Optional */
      solver.extract(model);
      if (solver.solve(IloGenerateBounds(*env,variables,prec))){
        Box boxaux;
        for (IloInt i=0; i<nvars; i++){
          IloNumVar mivar = variables[i];
```

```
          Interval iaux(solver.getMin(mivar), solver.getMax(mivar));
          boxaux.addInterval(iaux);
        }
        e_box.push_back(boxaux);
      }
      model.end();
    }
  }
  catch (IloException& ex){
    cerr<<"Error in Problem::eBox(): "<<ex<<endl;
    end();
    exit(1);
  }
}

void iBox(IloEnv *env){
  for (IloInt i=0; i<nsols;i++){
    for (IloInt j=0; j<nvars; j++){
      extendRightDomain(env, i, j);     // Right extend
      extendLeftDomain(env, i, j);      // Left extend
    }
  }
}

void storageOnlyDiffSolutions(){
  if (nsols > 1){
    for (IloInt i=0;i<nsols - 1;i++)
      for (IloInt j=i+1;j<nsols;j++)
        if (solutions[i] == solutions[j])
          solutions[j].setId(-1);

    for (IloInt i=nsols-1;i>=0;i--)
      if (solutions[i].getId()==-1){
        solutions.erase(solutions.begin() + i);
        i_box.erase(i_box.begin() + i);
        nsols--;
      }

    for (IloInt i=0;i<nsols;i++)
      solutions[i].setId(i);
  }
}

void solve(IloEnv *env, IloNum sprec=1e-8, bool nolineal=true){
  try {
    IloModel model(*env);
    for (IloInt i=0; i<ncons; i++){
      IloExpr exp = constraints[i].getExpr();
      model.add(exp == 0);
    }

    IloSolver solver(*env);
    if (nolineal)
      solver.useNonLinConstraint();
```

```
    solver.setDefaultPrecision(sprec);
    solver.extract(model);

    IlcSearch search = solver.newSearch(IloSplit(*env, variables));
    while (search.next()){
      Box ibox_aux(nvars);
      Solution aux(nsols++, nvars);
      aux.setBox(initdomain);
      for (IloInt i=0; i<nvars; i++){
        aux.setVarValue(i, solver.getValue(variables[i]));
        ibox_aux.setVarInterval(i, Interval(solver.getMin(variables[i]),
                                            solver.getMax(variables[i])));
      }
      solutions.push_back(aux);
      i_box.push_back(ibox_aux);

      if (nsols > 9999){
        cout<<"\nError: More than 9999 solutions found!. "<<endl;
        end();
        exit(1);
      }
    }
    search.end();
    solver.end();
    model.end();
  }
  catch (IloException& ex){
    cerr<<"Error in Problem::solve(): "<<ex<<endl;
    end();
    exit(1);
  }

  if (nsols){
    storageOnlyDiffSolutions();
    iBox(env);
    separateSols();
    eBox(env);
  }
}

void showSolutions(){
  if (nsols==0){
    cout<<"No solutions..."<<endl;
  } else {
    for (IloInt i=0; i<nsols; i++)
      showSol(i);
  }
}

void showSol(IloInt n){
  if (nsols>n){
    cout<<"Solution "<<n<<": ( "<<solutions[n].getVarValue(0);
    for (IloInt j=1; j<nvars; j++)
      cout<<"; "<<solutions[n].getVarValue(j);
```

```
    cout<<")"<<endl;
  }
}

void showeBox(IloInt n){
  if (nsols>n){
    cout<<"eBox Sol."<<n<<": ";
    for (IloInt j=0; j<nvars; j++)
      cout<<e_box[n].getVarInterval(j)<<" ";
    cout<<endl;
  }
}

void showiBox(IloInt n){
  if (nsols>n){
    cout<<"ibox Sol."<<n<<": ";
    for (IloInt j=0; j<nvars; j++)
      cout<<i_box[n].getVarInterval(j)<<" ";
    cout<<endl;
  }
}

void showDomain(IloInt n){
  if (nsols>n){
    cout<<"Doms Sol."<<n<<": ";
    for (IloInt j=0; j<nvars; j++)
      cout<<solutions[n].getVarInterval(j)<<" ";
    cout<<endl;
  }
}

void showAll(){
  if (nsols==0){
    cout<<"No solutions..."<<endl;
  } else {
    for (IloInt i=0; i<nsols; i++){
      cout<<endl;
      showSol(i);
      showDomain(i);
      showiBox(i);
      showeBox(i);
    }
  }
  cout<<endl;
}
```

```
  void end(){
    for (IloInt i=0; i<nvars; i++){
      Interval aux = initdomain.getVarInterval(i);
      variables[i].setBounds(aux.getLb(), aux.getUb());
    }
    constraints.clear();
    solutions.clear();
    initdomain.clear();
    i_box.clear();
    e_box.clear();
  }

  void addSpecialConstraint(IloConstraint cons){
    sconst.push_back(cons);
  }

};

//--------------------------
// Main Program: Sin and Log
// -=-=-=-=-=-=-=-=-=-=-=-=-=

int main(){
  IloEnv env;

  IloNumVar x(env, -9, 9);                  // Declaring variables
  IloNumVar y(env, 0, 200);                 // Declaring variables
  IloNumVarArray vars(env,2,x,y);           // Declaring array with ...
                                            // ...all variables for model
  Problem problem;                          // Declaring a "problem"
  problem.addVars(vars);                    // Informing about all variables

  problem.addConstraint(100*IloLog(x+10)*IloSin(2*x) - y -50, 0.3);
  problem.addConstraint(2*x*x - y, 0.2);

  problem.solve(&env);                      // solving :-)
  problem.showAll();
  problem.end();

  env.end();
  return 0;
}
```

Bibliografía

[AVA88] Jeffrey. D. Ullman Alfred. V. Aho, John. E. Hopcroft. *Estructura de Datos y Algoritmos*. Addison-Wesley Publishing Company, 1988.

[Bac98] Rolf Backofen. Constraint techniques for solving the protein structure prediction problem. *Lecture Notes in Computer Science*, 1520:72–88, 1998.

[BBM94] Renate Beckmann, Ulrich Bieker, and Ingolf Markhof. Application of constraint logic programming for VLSI CAD tools. In *Constraints in Computational Logics*, pages 183–200, 1994.

[BMV94] Frédéric Benhamou, David McAllester, and Pascal Van Hentenryck. CLP(intervals) revisited. In Maurice Bruynooghe, editor, *Proceedings of the 1994 International Symposium*, pages 124–138. MIT Press, 1994.

[Cha93] Philippe Charman. Solving space planning using constraint technology. In Manfred Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93, St. Petersburg, July 1993*, pages 159–172, DFKI Kaiserslautern, 1993.

[COP01] Projet COPRIN. Constraintes, optimisation, résolution par intervalles, 2001.

[Dan48] George Dantzig. Programming in a linear structure, 1948.

[Dec01] Dr. Rina Dechter. Constraint networks, 2001.

[Fal94] Boi Faltings. Arc-consistency for continuous variables. *Artificial Intelligence*, 65:363–376, 1994.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Co, 1979.

[Hen99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, 1999.

[Hél99] Hélène Collavizza, François Delobel, Michel Rueher. Extending consistent domains of numeric CSP. Technical report, Université de Nice-Sophia-Antipolis., 1999.

[ILO00a] ILOG. Ilog solver, reference manual, August 2000.

[ILO00b] ILOG. Ilog solver, user's manual, August 2000.

[KB99] Ludwig Krippahl and Pedro Barahona. Applying constraint programming to protein structure determination. In *Principles and Practice of Constraint Programming - CP'99, LNCS*, pages 289–302, 1999.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[Lau78] Jean-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[Lho93] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI '93, pages 232 238*, France, 1993. Chambry.

[LP01] Irvin J. Lustig and Jean-Francois Puget. Program does not equal program: Constraint programming and its relationship to mathematical programming. *Interfaces*, 31:29–53, December 2001.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mil03] Justin Miller. Arc consistency algorithms, ac-1, ac-2, ac-3, ac-4, 2003.

[Moo66] R. E. Moore. *Interval Analysis.* Prentice Hall, 1966.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints. An Introduction.* Massachusetts Institute of Technology, 1998.

[Nev02a] Bertrand Neveu. Concert/solver pour des systèmes de contraintes sur des réels, November 2002.

[Nev02b] Bertrand Neveu. Ilog concert - ilog solver. mini-manuel, October 2002.

[OV90] W. J. Older and A. Velino. Extending Prolog with constraint arithmetic on real intervals. In *Proc. of IEEE Canadian conference on Electrical and Computer Engineering*, New York, 1990. IEEE Computer Society Press.

[OV93] W. J. Older and A. Velino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. MIT Press, 1993.

[PVH99] Laurent Perron Jean-Charles Régin Pascal Van Hentenryck, Laurent Michel. Constraint programming in opl. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116, September 29 - October 1 1999.

[Rue02] Michel Rueher. Contraintes sur les domaines continus. Notes de cours (brouillon), December 2002.

[SSHF01] Marius-Călin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Search techniques for non-linear constraint satisfaction problems with inequalities. *Lecture Notes in Computer Science*, 2056:183–??, 2001.

[Tho93] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, Mark Wallace. Constraint logic programming. an informal introduction. Technical report, European Computer-Industry Research Center, February 1993.

[Tou02] Julien Touati. Traitement des incertitudes dans les contraintes de distance. Rapport de stage scientifique, ENPC, July 2002.

[ZW98] Yuanlin Zhang and Hui Wu. Bound consistency on linear constraints in finite domain constraint programming. In *ECAI 98. 13th European Conference on Artificial Inteligence*. John Wiley and Sons, Ltd., 1998.